Android

系统源代码情景分析（第三版）

◎罗升阳 著◎

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

在内容上，本书结合使用情景，全面、深入、细致地分析了Android系统的源代码，涉及Linux内核层、硬件抽象层（HAL）、运行时库层（Runtime）、应用程序框架层（Application Framework）及应用程序层（Application）。

在组织上，本书将上述内容划分为初识Android系统、Android专用驱动系统及Android应用程序框架三大篇。初识 Android系统篇介绍了参考书籍、基础知识及实验环境搭建；Android专用驱动系统篇介绍了Logger日志驱动程序、Binder进程间通信驱动程序及Ashmem匿名共享内存驱动程序；Android应用程序框架篇从组件、进程、消息和安装四个维度对Android应用程序的框架进行了深入的剖析。

通过上述内容及其组织，本书使读者既能从整体上把握Android系统的层次结构，又能从细节上掌握每一个层次的要点。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

Android系统源代码情景分析 / 罗升阳著. — 3版. —北京：电子工业出版社，2017.10
ISBN 978-7-121-32521-2

Ⅰ. ①A… Ⅱ. ①罗… Ⅲ. ①移动终端—应用程序—程序设计 Ⅳ. ①TN929.53

中国版本图书馆CIP数据核字（2017）第199928号

策划编辑：符隆美
责任编辑：葛　娜
印　　刷：北京京科印刷有限公司
装　　订：北京京科印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：850×1168　1/16　　印张：53　　字数：1585千字
版　　次：2012年10月第1版
　　2017年10月第3版
印　　次：2017年10月第1次印刷
定　　价：129.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前　言

Android系统自2008年9月发布第一个版本1.0以来，截至2016年8月发布最新版本7.0，一共存在十多个版本。本书在2012年10月出版第一版时，选择了Android 2.3的源代码来分析Android系统的实现，主要是因为就当时而言，它的基础架构是最稳定的，而且也是使用最广泛的。

本书有两个特点。一是通过使用情景来分析Android源代码。这种方式不仅能使读者带着目标去阅读源代码，还能把系统各个模块有机地联系在一起来理解。二是分析的内容都是Android系统的核心基础架构。这些内容在后来的版本中，都是灵魂般地稳定存在的。例如，智能指针、硬件抽象层（HAL）、匿名共享内存（Ashmem）、Binder IPC和四大组件管理等。

因此，读者阅读了本书之后，能够自行去阅读和分析其他的Android源代码。这种“授人以鱼，不如授人以渔”的效果，正是本书所追求的目标。因为技术发展日新月异，只有掌握了本质和宗旨，才能以不变应万变。

笔者记得2010年学习Linux内核时，一开始就从当时的最新版本（2.6.32）入手，结果犹如跌入茫茫大海之中。高版本的源代码，为了解决和优化一些问题，往往会引进一些复杂的概念，以及增加一些难以理解的代码。对于初学者来说，这些复杂的概念和难以理解的代码，会阻碍他们理解代码的核心思想。后来笔者改为从0.11版本（Linux的第一个基本可以正常运行的内核版本，于1991年发布）入手，才逐渐进入状态。《Linux内核完全注释》这本书，就是以0.11版本为蓝本，对Linux内核进行分析的。

本书自第一版发布以来，经历了6次重印，得到了读者的热烈反馈和建议。基于这些反馈和建议，本书在2015年12月进行了相应的修订。在不到两年的时间内，修订版又经历了4次重印，这充分说明了读者对本书的厚爱。本次修订仍以Android 2.3版本的源代码为蓝本，并且根据读者持续的反馈和建议（http://blog.csdn.net/luoshengyang/article/details/8116866）进行。由于读者提供的反馈和建议众多，这里就不一一列出，具体可以参考上述链接。

同时，本次修订还根据最新的官方文档，优化了Android源代码编译环境搭建的内容，使读者可以简单、快速、准确地初始化好Android源代码编译环境。另外，本次修订还增加了一节内容，介绍如何为真机编译ROM。让自己编译的Android源代码运行在真机上，是一件令人兴奋的事情，一方面，可以加深对Android系统的理解；另一方面，可以随心所欲地定制系统的行为，这不仅可以满足特殊的手机使用需求，还能辅助日常的开发工作。

为了更好地帮助读者理解本书的内容，笔者创建了一个网站（http://0xcc0xcd.com）。该网站会陆续提供一些免费视频，讲解Android系统相关的知识。最后，希望本书能够继续为有志于深入研究Android的读者服务。同时也非常感谢读者的反馈和建议，你们的支持是笔者持续研究和分享Android系统技术的动力所在。

本书内容

全书分为初识Android系统篇、Android专用驱动系统篇和Android应用程序框架篇三个部分。

初识Android系统篇包含三个章节的内容，主要介绍Android系统的基础知识。第1章介绍与Android系统有关的参考书籍、Android源代码工程环境的搭建方法，以及为真机编译ROM的过程；第2章介绍Android系统的硬件抽象层；第3章介绍Android系统的智能指针。读者可能会觉得奇怪，为什么一开始就介绍Android系统的硬件抽象层呢？因为涉及硬件，它似乎是一个深奥的知识点。其实不然，Android系统的硬件抽象层无论是从实现上，还是从使用上，它的层次都是非常清晰的；而且从下到上涵盖了整个Android系统，包括Android系统在用户空间和内核空间的实现。内核空间主要涉及硬件驱动程序的编写方法，而用户空间涉及运行时库层、应用程序框架层及应用程序层。因此，尽早学习Android系统的硬件抽象层，有助于我们从整体上去认识Android系统，以便后面可以更好地分析它的源代码。在分析Android系统源代码的过程中，经常会碰到智能指针，第3章我们就重点分析Android系统智能指针的实现原理，也是为了后面可以更好地分析Android系统源代码。

Android专用驱动系统篇包含三个章节的内容。我们知道，Android系统是基于Linux内核来开发的，但是由于移动设备的CPU和内存配置都要比PC低，因此，Android系统并不是完全在Linux内核上开发的，而是在Linux内核里面添加了一些专用的驱动模块来使它更适合于移动设备。这些专用的驱动模块同时也形成了Android系统的坚实基础，尤其是Logger日志驱动程序、Binder进程间通信驱动程序，以及Ashmem匿名共享内存驱动程序，它们在Android系统中被广泛地使用。在此篇中，我们分别在第4章、第5章和第6章分析Logger日志系统、Binder进程间通信系统和Ashmem共享内存系统的实现原理，为后面深入分析Android应用程序的框架打下良好的基础。

Android应用程序框架篇包含十个章节的内容。我们知道，在移动平台中，Android系统和iOS系统比的是谁的应用程序更丰富、质量更高、用户体验更好，谁就能取得最终的胜利。因此，每个平台都在尽最大努力吸引第三方开发者来为其开发应用程序。这就要求平台必须提供良好的应用程序架构，以便第三方开发者可以将更多的精力集中在应用程序的业务逻辑上，从而开发出数量更多、质量更高和用户体验更好的应用程序。在此篇中，我们将从组件、进程、消息和安装四个维度来分析Android应用程序的实现框架。第7章到第10章分析Android应用程序四大组件Activity、Service、Broadcast Receiver和Content Provider的实现原理；第11章和第12章分析Android应用程序进程的启动过程；第13 章到第15章分析Android应用程序的消息处理机制；第16章分析Android应用程序的安装和显示过程。学习了这些知识之后，我们就可以掌握Android系统的精髓了。

本书特点

本书从初学者的角度出发，结合具体的使用情景，在纵向和横向上对Android系统的源代码进行了全面、深入、细致的分析。在纵向上，采用从下到上的方式，分析的源代码涉及Android系统的内核层（Linux Kernel）、硬件抽象层（HAL）、运行时库层（Runtime）、应用程序框架层（Application Framework）以及应用程序层（Application），这有利于读者从整体上掌握Android系统的架构。在横向上，从Android应用程序的组件、进程、消息和安装四个维度出发，全面地剖析Android系统的应用程序框架层，这有利于读者深入理解Android应用程序的架构及运行原理。

目　　录

第1篇　初识Android系统

第1章　准备知识 2

1.1　Linux内核参考书籍 2
1.2　Android应用程序参考书籍 3
1.3　下载、编译和运行Android源代码 3
1.3.1　下载Android源代码 5
1.3.2　编译Android源代码 6
1.3.3　运行Android模拟器 7
1.4　下载、编译和运行Android内核源代码 8
1.4.1　下载Android内核源代码 8
1.4.2　编译Android内核源代码 8
1.4.3　运行Android模拟器 9
1.5　开发第一个Android应用程序 10
1.6　单独编译和打包Android应用程序模块 12
1.6.1　导入单独编译模块的mmm命令 12
1.6.2　单独编译Android应用程序模块 13
1.6.3　重新打包Android系统镜像文件 13
1.7　为真机编译ROM 14
1.7.1　下载LineageOS 16
1.7.2　下载设备开源代码 16
1.7.3　下载设备私有文件 17
1.7.4　编译ROM 17
1.7.5　刷入TWRP 17
1.7.6　刷入ROM 19

第2章　硬件抽象层 21

2.1　开发Android硬件驱动程序 22
2.1.1　实现内核驱动程序模块 22
2.1.2　修改内核Kconfig文件 29

2.1.3 修改内核Makefile文件 30
2.1.4 编译内核驱动程序模块 30
2.1.5 验证内核驱动程序模块 31
2.2 开发C可执行程序验证Android硬件驱动程序 32
2.3 开发Android硬件抽象层模块 34
2.3.1 硬件抽象层模块编写规范 34
2.3.2 编写硬件抽象层模块接口 37
2.3.3 硬件抽象层模块的加载过程 41
2.3.4 处理硬件设备访问权限问题 44
2.4 开发Android硬件访问服务 46
2.4.1 定义硬件访问服务接口 46
2.4.2 实现硬件访问服务 47
2.4.3 实现硬件访问服务的JNI方法 48
2.4.4 启动硬件访问服务 51
2.5 开发Android应用程序来使用硬件访问服务 52

第3章 智能指针 57
3.1 轻量级指针 58
3.1.1 实现原理分析 58
3.1.2 应用实例分析 61
3.2 强指针和弱指针 62
3.2.1 强指针的实现原理分析 63
3.2.2 弱指针的实现原理分析 69
3.2.3 应用实例分析 75

第2篇 Android专用驱动系统

第4章 Logger日志系统 82
4.1 Logger日志格式 83
4.2 Logger日志驱动程序 84
4.2.1 基础数据结构 85
4.2.2 日志设备的初始化过程 86
4.2.3 日志设备文件的打开过程 91
4.2.4 日志记录的读取过程 92
4.2.5 日志记录的写入过程 96
4.3 运行时库层日志库 101
4.4 C/C++日志写入接口 108
4.5 Java日志写入接口 112
4.6 Logcat工具分析 118
4.6.1 基础数据结构 119

4.6.2 初始化过程 123
4.6.3 日志记录的读取过程 135
4.6.4 日志记录的输出过程 140

第5章 Binder进程间通信系统 152

5.1 Binder驱动程序 153
5.1.1 基础数据结构 154
5.1.2 Binder设备的初始化过程 172
5.1.3 Binder设备文件的打开过程 173
5.1.4 Binder设备文件的内存映射过程 174
5.1.5 内核缓冲区管理 181
5.2 Binder进程间通信库 191
5.3 Binder进程间通信应用实例 196
5.4 Binder对象引用计数技术 204
5.4.1 Binder本地对象的生命周期 205
5.4.2 Binder实体对象的生命周期 209
5.4.3 Binder引用对象的生命周期 212
5.4.4 Binder代理对象的生命周期 217
5.5 Binder对象死亡通知机制 220
5.5.1 注册死亡接收通知 221
5.5.2 发送死亡接收通知 224
5.5.3 注销死亡接收通知 229
5.6 Service Manager的启动过程 232
5.6.1 打开和映射Binder设备文件 234
5.6.2 注册为Binder上下文管理者 235
5.6.3 循环等待Client进程请求 239
5.7 Service Manager代理对象的获取过程 246
5.8 Service组件的启动过程 252
5.8.1 注册Service组件 253
5.8.2 启动Binder线程池 297
5.9 Service代理对象的获取过程 299
5.10 Binder进程间通信机制的Java接口 308
5.10.1 Service Manager的Java代理对象的获取过程 308
5.10.2 Java服务接口的定义和解析 318
5.10.3 Java服务的启动过程 321
5.10.4 Java服务代理对象的获取过程 328
5.10.5 Java服务的调用过程 331

第6章 Ashmem匿名共享内存系统 335

6.1 Ashmem驱动程序 336
6.1.1 基础数据结构 336
6.1.2 匿名共享内存设备的初始化过程 338

6.1.3 匿名共享内存设备文件的打开过程 340
6.1.4 匿名共享内存设备文件的内存映射过程 342
6.1.5 匿名共享内存块的锁定和解锁过程 344
6.1.6 匿名共享内存块的回收过程 352
6.2 运行时库cutils的匿名共享内存访问接口 353
6.3 匿名共享内存的C++访问接口 357
6.3.1 MemoryHeapBase 357
6.3.2 MemoryBase 367
6.3.3 应用实例 372
6.4 匿名共享内存的Java访问接口 378
6.4.1 MemoryFile 378
6.4.2 应用实例 383
6.5 匿名共享内存的共享原理 394

第3篇 Android应用程序框架

第7章 Activity组件的启动过程 400
7.1 Activity组件应用实例 400
7.2 根Activity组件的启动过程 406
7.3 子Activity组件在进程内的启动过程 440
7.4 子Activity组件在新进程中的启动过程 448

第8章 Service组件的启动过程 451
8.1 Service组件应用实例 451
8.2 Service组件在新进程中的启动过程 459
8.3 Service组件在进程内的绑定过程 471

第9章 Android系统广播机制 494
9.1 广播机制应用实例 495
9.2 广播接收者的注册过程 501
9.3 广播的发送过程 509

第10章 Content Provider组件的实现原理 532
10.1 Content Provider组件应用实例 533
10.1.1 ArticlesProvider 533
10.1.2 Article 543
10.2 Content Provider组件的启动过程 558
10.3 Content Provider组件的数据共享原理 581

10.3.1 数据共享模型 581
10.3.2 数据传输过程 584
10.4 Content Provider组件的数据更新通知机制 604
10.4.1 注册内容观察者 605
10.4.2 发送数据更新通知 611

第11章 Zygote和System进程的启动过程 619

11.1 Zygote进程的启动脚本 619
11.2 Zygote进程的启动过程 622
11.3 System进程的启动过程 630

第12章 Android应用程序进程的启动过程 638

12.1 应用程序进程的创建过程 638
12.2 Binder线程池的启动过程 647
12.3 消息循环的创建过程 649

第13章 Android应用程序的消息处理机制 653

13.1 创建线程消息队列 653
13.2 线程消息循环过程 658
13.3 线程消息发送过程 663
13.4 线程消息处理过程 668

第14章 Android应用程序的键盘消息处理机制 675

14.1 键盘消息处理模型 675
14.2 InputManager的启动过程 678
14.2.1 创建InputManager 678
14.2.2 启动InputManager 681
14.2.3 启动InputDispatcher 683
14.2.4 启动InputReader 685
14.3 InputChannel的注册过程 696
14.3.1 创建InputChannel 697
14.3.2 注册Server端InputChannel 705
14.3.3 注册系统当前激活的应用程序窗口 709
14.3.4 注册Client端InputChannel 714
14.4 键盘消息的分发过程 717
14.4.1 InputReader获得键盘事件 718
14.4.2 InputDispatcher分发键盘事件 725
14.4.3 系统当前激活的应用程序窗口获得键盘消息 735
14.4.4 InputDispatcher获得键盘事件处理完成通知 751

14.5 InputChannel的注销过程 754
14.5.1 销毁应用程序窗口 755
14.5.2 注销Client端InputChannel 764
14.5.3 注销Server端InputChannel 766

第15章 Android应用程序线程的消息循环模型 772

15.1 应用程序主线程消息循环模型 773
15.2 与界面无关的应用程序子线程消息循环模型 774
15.3 与界面相关的应用程序子线程消息循环模型 777

第16章 Android应用程序的安装和显示过程 786

16.1 应用程序的安装过程 786
16.2 应用程序的显示过程 822

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源：**本书所提供的示例代码及资源文件，均可在下载资源处下载。
- **提交勘误：**您对书中内容的修改意见可在提交勘误处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方读者评论处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/32521



第1篇　初识Android系统

Linux内核的鼻祖Linus在回答一个Minix系统[1]的问题时，第一句话便是“Read The Fucking Source Code”。这句话虽然颇有调侃的味道，但是它道出了阅读源代码的重要性。由于Android系统的源代码是开放的，因此，认识Android系统的最好方法莫过于阅读它的源代码。

我们知道，Android系统是基于Linux内核来开发的，因此，在阅读它的源代码之前，需要掌握Linux内核的基础知识。有了Linux内核的基础知识之后，就可以下载Android源代码来阅读。阅读Android源代码时，宜采用动静结合的方法。所谓静，就是研究和思考源代码的实现；而所谓动，就是通过运行系统来证实自己对源代码的研究和思考。运行系统并不是单纯地将系统运行起来就可以了，还需要亲自动手编写一些应用程序来验证系统的行为。因此，在第1章中，我们首先介绍几本Linux内核的经典参考书籍，然后介绍如何搭建Android源代码工程环境，最后介绍如何在Android源代码工程环境中开发应用程序。

在阅读Android源代码之前，首先需要从整体上了解Android系统的架构。Android系统大致可以划分为五个层次，它们分别是Linux内核层、硬件抽象层、运行时库层、应用程序框架层和应用程序层。其中，硬件抽象层从实现到使用上涉及这五个层次，因此，在第2章中，我们通过具体的实例来介绍Android系统的硬件抽象层，以便对Android系统的实现层次有一个感性的认识。

虽然Android应用程序是使用Java语言来开发的，但是在Android应用程序框架层中，有相当多的一部分代码使用C++语言来编写，在阅读这些C++代码时，经常会碰到智能指针。Android系统的智能指针对于C++开发者来说，是比较容易理解的；但是对于Java开发者来说，就比较苦涩了。因此，在第3章中，我们继续通过具体的实例来分析Android系统的智能指针实现原理。

1　Minix是一个类UNIX的操作系统，见http://en.wikipedia.org/wiki/MINIX。

第 1 章
准 备 知 识

Android系统的源代码非常庞大和复杂，我们不能贸然进入，否则很容易在里面迷失方向，进而失去研究它的信心。为了有条不紊地对Android系统的源代码进行全面、深入、细致的分析，我们需要准备一些参考书籍，搭建好Android系统源代码工程环境，以及对Android系统有一个感性认识。

1.1 Linux内核参考书籍

在阅读分析Android系统的源代码时，经常会碰到诸如管道（pipe）、套接字（socket）和虚拟文件系统（VFS）等知识。此外，Android系统通过模块的形式在Linux内核中增加了一些专用的驱动程序，如Logger日志驱动程序、Binder进程间通信驱动程序，以及Ashmem匿名共享内存驱动程序等，这些都是Linux内核的基础知识，涉及进程、内存管理等内容。由于本书的重点是分析Android系统的源代码，因此，下面推荐四本介绍Linux内核基础知识的经典书籍。

（1）*Linux Kernel Development*

这本书的作者是Robert Love，目前最新的版本是第3版。它对Linux内核的设计原理和实现思路提供了一个总览视图，并且对Linux内核的各个子系统的设计目标进行了清晰的描述，非常适合初学者阅读。从软件工程的角度来看，这本书相当于Linux内核的概要设计文档。

（2）*Understanding the Linux Kernel*

这本书的作者是Daniel P. Bovet和Marco Cesati，目前最新的版本是第3版。它对Linux内核的实现提供了更多的细节，详细地描述了内核开发中用到的各种重要数据结构、算法以及编程技巧等，非常适合中、高级读者阅读。从软件工程的角度来看，这本书相当于Linux内核的详细设计文档。

（3）*Linux Device Drivers*

这本书的作者是Jonathan Corbet, Alessandro Rubini和Greg Kroah-Hartman，目前最新的版本是第3版。它更加注重于实际操作，详细地讲解了Linux内核驱动程序的实现原理，对分析Android系统的专用驱动模块有非常大的帮助。

（4）《*Linux*内核源代码情景分析》

这本书的作者是毛德操和胡希明，是中国人自己编写的一本经典的Linux内核书籍。它最大的特点是从使用情景出发，对Linux内核作了详细的分析，为读者在Linux内核源代码的汪洋大海中指明方向。

1.2 Android应用程序参考书籍

分析Android系统的源代码时，应该带着问题或者目标。要把问题或者目标挖掘出来，最好的方法就是在Android系统中编写应用程序。通过编写Android应用程序，我们可以知道系统提供了哪些功能，并且如何去使用这些功能，进而激发我们去了解这些功能是如何实现的。这样我们就可以获得分析Android系统源代码所需要的问题或者目标。下面推荐两本介绍Android应用程序开发的书籍。

（1）*Professional Android 2 Application Development*

（2）《*Google Android SDK*开发范例大全》

这两本书的特点是都使用了大量的例子来描述如何使用Android SDK来开发Android应用程序。我们可以根据实际情况来熟悉Android应用程序的开发方法，主要掌握Android应用程序四大组件（Activity、Service、Broadcast Receiver和Content Provider）的用法。在学习的过程中，如果遇到其他问题，还可以参考官方API文档，其网址为：

http://developer.android.com/index.html

1.3 下载、编译和运行Android源代码

目前，官方支持Android源代码在Ubuntu和Mac系统上编译。本书推荐使用Ubuntu系统。Ubuntu系统是一个广受称道的Linux发行版本，它具有强大的软件包管理系统，并且简单易用，官方下载地址为：http://www.ubuntu.com/。

不同版本的Android源代码对Ubuntu系统版本的要求不一样，具体如下。

- Android 6.0 (Marshmallow) ~ AOSP master：Ubuntu 14.04 (Trusty)
- Android 2.3.x (Gingerbread) ~ Android 5.x (Lollipop)：Ubuntu 12.04 (Precise)
- Android 1.5 (Cupcake) ~ Android 2.2.x (Froyo)：Ubuntu 10.04 (Lucid)

本书分析的Android源代码版本为2.3.x，因此推荐安装Ubuntu 12.04。另外，如果要安装Ubuntu 14.04，则需要安装64位版本。

如果读者习惯使用Windows系统，就可以考虑先在Windows上安装虚拟机，然后在虚拟机上安装Ubuntu系统。虚拟机推荐使用VMware，官方网站为：http://www.vmware.com/。

安装VMware时，最好选择6.0以上的版本，因为较旧版本的VMware在网络连接支持上比较差，而我们在下载Android源代码时，是必须要联网的。

安装好Ubuntu系统之后，还需要安装JDK及其他的依赖包，然后才可以正常下载、编译和运行Android源代码。接下来我们就介绍它们的安装方法[1]。

1. JDK

不同版本的Android源代码对JDK版本的要求也不一样，具体如下。

- AOSP master：OpenJDK 8
- Android 5.x (Lollipop) ~ Android 6.0 (Marshmallow)：OpenJDK 7
- Android 2.3.x (Gingerbread) ~ Android 4.4.x (KitKat)：Java JDK 6
- Android 1.5 (Cupcake) ~ Android 2.2.x (Froyo)：Java JDK 5

1 随着Android源代码版本的迭代，对于Ubuntu版本、JDK及其他依赖包都会发生变化，最新的安装指令可以参考官方网站：https://source.android.com/source/initializing。

Java JDK 5和6的官方网站为：http://www.oracle.com。

该网站为Linux提供的JDK有bin和rpm.bin两种包，推荐下载bin包。以JDK 6为例，假设下载文件为jdk-6u45-linux-x64.bin，保存在~/Downloads目录下，执行以下命令进行安装。

```
USER@MACHINE:~$ sudo cp ~/Downloads/jdk-6u45-linux-x64.bin /usr/java/
USER@MACHINE:~$ cd /usr/java
USER@MACHINE:/usr/java$ chmod a+x ./jdk-6u45-linux-x64.bin
USER@MACHINE:/usr/java$ ./jdk-6u45-linux-x64.bin
```

安装完成后，可以通过update-alternatives工具管理JDK，如下所示。

```
USER@MACHINE:~$ sudo update-alternatives -install /usr/bin/java java /usr/java/jdk1.6.0_45/bin/
java 1061
USER@MACHINE:~$ sudo update-alternatives -install /usr/bin/javac javac /usr/java/jdk1.6.0_45/
bin/java 1061
```

如果电脑上也安装了其他版本的JDK，则可以通过以下命令选择所需要的版本。

```
USER@MACHINE:~$ sudo update-alternatives -config java
There are 5 choices for the alternative java (providing /usr/bin/java).
  Selection    Path                                            Priority   Status
------------------------------------------------------------------------------
  0            /usr/java/jdk1.8.0_131/jre/bin/java              1082      auto mode
  1            /usr/java/jdk1.6.0_45/bin/java                   1061      manual mode
* 2            /usr/java/jdk1.8.0_131/bin/java                  1080      manual mode
  3            /usr/java/jdk1.8.0_131/jre/bin/java              1082      manual mode
  4            /usr/lib/jvm/java-7-openjdk-amd64/jre/bin/java   1071      manual mode
  5            /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java   1081      manual mode
Press <enter> to keep the current choice[*], or type selection number: 1
```

输入数字“1”，就可以将系统的java命令指向JDK 6安装目录下的Java文件。

```
USER@MACHINE:~$ sudo update-alternatives -config javac
There are 4 choices for the alternative javac (providing /usr/bin/javac).
  Selection    Path                                          Priority   Status
------------------------------------------------------------------------------
  0            /usr/lib/jvm/java-8-openjdk-amd64/bin/javac   1081      auto mode
  1            /usr/java/jdk1.6.0_45/bin/javac               1061      manual mode
* 2            /usr/java/jdk1.8.0_131/bin/javac              1080      manual mode
  3            /usr/lib/jvm/java-7-openjdk-amd64/bin/javac   1071      manual mode
  4            /usr/lib/jvm/java-8-openjdk-amd64/bin/javac   1081      manual mode
Press <enter> to keep the current choice[*], or type selection number: 1
```

输入数字“1”，就可以将系统的javac命令指向JDK 6安装目录下的javac文件。

OpenJDK 7和8的下载地址分别如下：

http://archive.ubuntu.com/ubuntu/pool/universe/o/openjdk-7/

http://archive.ubuntu.com/ubuntu/pool/universe/o/openjdk-8/

下载得到一个deb文件，可以使用dpkg命令进行安装。例如，假设下载OpenJDK 8，文件名为openjdk-8-dbg_8u45-b14-1_amd64.deb，保存在~/Downloads目录下，执行以下命令进行安装。

```
USER@MACHINE:~$ sudo apt-get update
USER@MACHINE:~$ sudo dpkg -i ~/Downloads/openjdk-8-dbg_8u45-b14-1_amd64.deb
```

在安装过程中如果提示依赖缺失，则可以执行以下命令进行修复安装。

```
USER@MACHINE:~$ sudo apt-get -f install
```

2. 其他依赖包

Ubuntu 14.04的依赖包可以执行以下命令进行安装。

```
USER@MACHINE:~$ sudo apt-get install git-core gnupg flex bison gperf build-essential \
  zip curl zlib1g-dev gcc-multilib g++-multilib libc6-dev-i386 \
  lib32ncurses5-dev x11proto-core-dev libx11-dev lib32z-dev ccache \
  libgl1-mesa-dev libxml2-utils xsltproc unzip
```

Ubuntu 12.04的依赖包可以执行以下命令进行安装。

```
USER@MACHINE:~$ sudo apt-get install git gnupg flex bison gperf build-essential \
  zip curl libc6-dev libncurses5-dev:i386 x11proto-core-dev \
  libx11-dev:i386 libreadline6-dev:i386 libgl1-mesa-glx:i386 \
  libgl1-mesa-dev g++-multilib mingw32 tofrodos \
  python-markdown libxml2-utils xsltproc zlib1g-dev:i386
USER@MACHINE:~$ sudo ln -s /usr/lib/i386-linux-gnu/mesa/libGL.so.1 /usr/lib/i386-linux-gnu/
libGL.so
```

Ubuntu 10.04的依赖包可以执行以下命令进行安装。

```
USER@MACHINE:~$ sudo apt-get install git gnupg flex bison gperf build-essential \
  zip curl zlib1g-dev libc6-dev lib32ncurses5-dev ia32-libs \
  x11proto-core-dev libx11-dev lib32readline5-dev lib32z-dev \
  libgl1-mesa-dev g++-multilib mingw32 tofrodos python-markdown \
  libxml2-utils xsltproc
```

在上述依赖包中，包含了Git工具。Git是一种分布式的源代码管理工具，它可以有效、高速地对项目源代码进行版本管理[1]。Android源代码就是采用它来管理的。

1.3.1 下载Android源代码

为了方便开发者下载Android源代码，Google提供了一个repo工具。这个工具实际上是一个脚本文件，里面封装了用来下载Android源代码所需要的git命令。它的下载和安装方法如下：

```
USER@MACHINE:~$ wget https://dl-ssl.google.com/dl/googlesource/git-repo/repo
USER@MACHINE:~$ chmod a+x repo
USER@MACHINE:~$ sudo mv repo /bin/
```

安装好repo工具之后，我们就可以创建一个空目录，然后进入到这个目录中执行repo命令来下载Android源代码了。

```
USER@MACHINE:~$ mkdir Android
USER@MACHINE:~$ cd Android
USER@MACHINE:~/Android$ repo init -u https://android.googlesource.com/platform/manifest
USER@MACHINE:~/Android$ repo sync
```

下载的过程可能会比较漫长，这取决于网络连接速度，期间还可能会碰到网络中断的现象，这时候只需要重复执行repo sync命令就可以继续下载了。

上述命令下载的是主线上的Android源代码，即最新版本的Android源代码。一般来说，主线上的源代码是正在开发的版本，它是不稳定的，编译和运行时都可能会遇到问题。如果想下载稳定的版本，就需要选择某一个支线上的代码。例如，如果我们想下载Android 2.3.1版本的代码，就可以在执行repo init命令时指定-b选项。

```
USER@MACHINE:~/Android$ repo init -u https://android.googlesource.com/platform/manifest -b
android-2.3.1_r1
```

在本书接下来的内容中，如果没有特别声明，我们所分析的Android源代码都是基于Android 2.3版本的，并且位于~/Android目录中。

1 Git工具的使用方法可以参考官方网站：http://git-scm.com/。

1.3.2 编译Android源代码

要编译Android源代码，只需在Android源代码目录下执行make命令就可以了。

```
USER@MACHINE:~/Android$ make
```

第一次编译Android源代码时，花费的时间会比较长，同时也可能会遇到各种各样的问题，这时候一般都可以通过搜索引擎来找到解决方案。例如，如果我们是在32位机器上编译主线上的Android源代码，则会碰到下面这个错误提示。

```
build/core/main.mk:76: ************************************************************
build/core/main.mk:77: You are attempting to build on a 32-bit system.
build/core/main.mk:78: Only 64-bit build environments are supported beyond froyo/2.2.
build/core/main.mk:79: ************************************************************
```

这时候可以使用关键词“You are attempting to build on a 32-bit system”在搜索引擎上找到解决方案。原来，主线上的Android源代码默认只能在64位的机器上编译，如果在32位的机器上编译，就会出现上述错误提示。如果我们仍然想在32位的机器上编译Android源代码，就可以按照下面方法来修改编译脚本。

（1）打开build/core/main.mk文件，并且找到下面内容。

```
ifeq ($(BUILD_OS),linux)
     build_arch := $(shell uname -m)
     ifneq (64,$(findstring 64,$(build_arch)))
     $(warning ********************************************************)
     $(warning You are attempting to build on a 32-bit system.)
     $(warning Only 64-bit build environments are supported beyond froyo/2.2.)
```

将第3行修改为

```
ifneq (i686,$(findstring i686,$(build_arch)))
```

（2）打开external/clearsilver/cgi/Android.mk、external/clearsilver/cs/Android.mk、external/clearsilver/java-jni/Android.mk和external/clearsilver/util/Android.mk这四个文件，并且找到下面内容：

```
# This forces a 64-bit build for Java6
LOCAL_CFLAGS += -m64
LOCAL_LDFLAGS += -m64
```

将后面两行修改为

```
LOCAL_CFLAGS += -m32
LOCAL_LDFLAGS += -m32
```

经过这样的修改之后，在32位的机器上编译Android源代码产生的问题就可以解决了。

编译成功后，可以看到下面的输出。

```
Target system fs image: out/target/product/generic/obj/PACKAGING/systemimage_intermediates/
system.img
Install system fs image: out/target/product/generic/system.img
Target ram disk: out/target/product/generic/ramdisk.img
Target userdata fs image: out/target/product/generic/userdata.img
Installed file list: out/target/product/generic/installed-files.txt
```

编译结果输出目录为out/target/product/$(TARGET_PRODUCT)，其中，TARGET_PRODUCT是一个

环境变量，它的默认值为generic。

Android源代码编译成功之后，可以执行以下命令将它打包成SDK。

```
USER@MACHINE:~/Android$ make sdk
```

打包成功后，可以看到下面的输出。

```
Package SDK: out/host/linux-x86/sdk/android-sdk_eng.$USER$_linux-x86.zip
```

其中，$USER$表示当前登录到系统中的用户名。有了这个SDK包之后，我们就可以在IDE环境中开发Android应用程序了。

1.3.3 运行Android模拟器

Android源代码编译成功之后，我们就可以运行它。为了方便起见，我们使用Android模拟器emulator来运行编译出来的Android源代码。执行以下命令来启动Android模拟器。

```
USER@MACHINE:~/Android$ export PATH=$PATH:~/Android/out/host/linux-x86/bin
USER@MACHINE:~/Android$ export ANDROID_PRODUCT_OUT=~/Android/out/target/product/generic
USER@MACHINE:~/Android$ emulator
```

Android模拟器位于源代码根目录下的子目录out/host/linux-x86/bin中，因此，为了方便使用，我们把这个目录添加到环境变量PATH中。

启动Android模拟器需要四个文件，它们分别是zImage、system.img、userdata.img和ramdisk.img，其中，前面一个是Linux内核镜像文件，而后面三个是Android系统镜像文件。如果不带任何参数来运行emulator命令，那么Android模拟器默认使用的zImage文件是位于源代码根目录下的子目录out/host/linux-x86/bin中的kernel-qemu文件，而默认使用的system.img、userdata.img和ramdisk.img文件则位于ANDROID_PRODUCT_OUT目录中。ANDROID_PRODUCT_OUT是一个环境变量，我们将它的值设置为Android源代码编译结果输出目录；如果不设置ANDROID_PRODUCT_OUT环境变量，就需要指定上述四个文件来启动Android模拟器，如下所示。

```
USER@MACHINE:~/Android$ emulator -kernel ./prebuilt/android-arm/kernel/kernel-qemu -sysdir ./
out/target/product/generic -system system.img -data userdata.img -ramdisk ramdisk.img
```

在本书接下来的内容中，当我们运行不带参数的emulator命令时，如果没有特别声明，都是假设已经设置好PATH和ANDROID_PRODUCT_OUT两个环境变量，其中PATH环境变量包含了~/Android/out/host/linux-x86/bin目录，而ANDROID_PRODUCT_OUT环境变量指向~/Android/out/target/product/generic目录。

Android模拟器emulator启动起来后，它的界面如图1-1所示。



图1-1 Android模拟器界面

1.4 下载、编译和运行Android内核源代码

Android源代码工程默认是不包含它所使用的Linux内核的源代码的，因此，如果我们需要运行定制的Linux内核，就要下载它的源代码，并且对它进行编译。接下来，我们就详细介绍如何为Android模拟器下载、编译和运行Linux内核源代码。

1.4.1 下载Android内核源代码

Android模拟器所使用的Linux内核源代码也是通过Git工具来管理的，不过它不像Android源代码有一个repo脚本工具来支持下载，我们需要手动执行git命令来下载。

```
USER@MACHINE:~/Android$ mkdir kernel
USER@MACHINE:~/Android$ cd kernel
USER@MACHINE:~/Android/kernel$ git clone http://android.googlesource.com/kernel/goldfish.git
```

首先在Android源代码根目录下创建一个用来保存Linux内核源代码的子目录kernel，然后调用git clone命令来下载Linux内核源代码。取决于网络连接速度，整个下载过程可能需要较长时间的等待。

下载完成之后，在kernel目录下就会看到一个空的goldfish子目录，这时候需要执行git checkout命令来指定所需要的支线代码。

```
USER@MACHINE:~/Android/kernel/goldfish$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/android-goldfish-2.6.29
  remotes/origin/master
USER@MACHINE:~/Android/kernel/goldfish$ git checkout remotes/origin/android-goldfish-2.6.29
```

在执行git checkout命令之前，可以先执行git branch命令来查看有哪些支线代码。由于我们使用Android模拟器来运行Android系统，因此，我们选择android-goldfish-2.6.29支线代码来作为Android系统的内核源代码。

执行完成git checkout命令之后，我们就可以在goldfish目录中看到Linux内核源代码了。在本书接下来的内容中，如果没有特别声明，我们分析的Android内核源代码都是基于android-goldfish-2.6.29版本的，并且位于~/Android/kernel/goldfish目录中。

1.4.2 编译Android内核源代码

在编译Android内核源代码之前，我们首先要修改它的Makefile文件。Android模拟器所使用的CPU体系结构是arm的，因此，我们需要将Makefile文件中ARCH变量的值设置为arm。又由于我们是在PC上为Android模拟器编译内核的，因此，还需要在Makefile文件中指定交叉编译工具，即修改里面的CROSS_COMPILE变量的值。

打开Android内核源代码目录下的Makefile文件，并且找到下面的内容。

```
export KBUILD_BUILDHOST := $(SUBARCH)
ARCH                ?= $(SUBARCH)
CROSS_COMPILE       ?=
```

将后面两行修改为

```
ARCH                ?= arm
CROSS_COMPILE       ?= arm-eabi-
```

Android源代码目录为我们准备了一个适用于编译Android模拟器内核的交叉编译工具，它位于Android源代码目录下的prebuilt/linux-x86/toolchain子目录中。在Makefile文件中，我们将ARCH变量的值设置为arm，表示编译的Linux内核是适用于arm体系结构的；而将CROSS_COMPILE变量的值设置为arm-eabi-，表示所使用的交叉编译工具名称是以“arm-eabi-”来作为前缀的。

为Android模拟器编译内核分为三个步骤。其中，第一步是将交叉编译工具所在的目录添加到环境变量PATH中；第二步是修改硬件配置文件goldfish_defconfig[1]；第三步是执行make命令。

```
USER@MACHINE:~/Android/kernel/goldfish$ export
PATH=$PATH:~/Android/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin
USER@MACHINE:~/Android/kernel/goldfish$ make goldfish_defconfig
USER@MACHINE:~/Android/kernel/goldfish$ make
```

编译成功后，可以看到下面两行输出。

```
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
```

这表明编译好的Android内核镜像文件zImage位于Android内核源代码目录下的arch/arm/boot子目录中。

1.4.3 运行Android模拟器

在前面1.3.3小节提到，在执行不带参数的emulator命令来启动Android模拟器时，使用的内核镜像文件zImage是位于Android源代码根目录下的out/host/linux-x86/bin子目录中的kernel-qemu文件，现在既然已经编译了自己的内核镜像文件zImage，我们就需要通过-kernel选项来启动Android模拟器，以便可以指定Android模拟器所使用的内核镜像文件。

```
USER@MACHINE:~/Android$ emulator -kernel ./kernel/goldfish/arch/arm/boot/zImage &
```

在命令后面加一个符号“&”，表示要在后台启动Android模拟器。这样做的目的是为了能够在同一个终端运行adb[2]工具来连接Android模拟器，以便验证Android模拟器是否使用了我们自己编译的Android内核镜像文件。

Android源代码目录中包含了adb工具，它位于~/Android/out/host/linux-x86/bin目录中。我们通过执行以下命令来查看Android模拟器所使用的内核镜像文件的信息。

```
USER@MACHINE:~/Android$ adb shell
root@android:/ # cd proc
root@android:/proc # cat version
Linux version 2.6.29-g46b05b2-dirty (USER@MACHINE) (gcc version 4.4.3 (GCC) ) #1 Sat Nov 26
15:27:21 HKT 2011
```

从最后两行就可以看出，Android模拟器使用的内核镜像文件的版本为Linux version 2.6.29，编译主机名为USER@MACHINE，编译工具为gcc version 4.4.3，以及编译日期为Sat Nov 26 15:27:21 HKT 2011。其中，USER@MACHINE和Sat Nov 26 15:27:21 HKT 2011对应于编译该内核镜像文件时所使用的主机名和时间，通过它们就可以验证Android模拟器所使用的内核镜像文件是否是我们自己编译出来

1 取决于Android模拟器的CPU体系架构版本，如果是arm v5版本的，则使用goldfish_defconfig文件；如果是arm v7版本的，则使用goldfish_armv7_defconfig文件。可以使用Android源代码目录下的prebuilt/android-arm/kernel子目录中的kernel-qemu文件来启动Android模拟器，然后使用adb工具来连接Android模拟器，并且通过cat命令来查看/proc/cpuinfo文件的内容，就可以知道当前所使用的Android模拟器的CPU体系架构版本。

2 Android Debug Bridge，这是一个用来调试Android系统的工具，它的使用方法可以参考官方文档http://developer.android.com/guide/developing/tools/adb.html。

的zImage文件。

1.5 开发第一个Android应用程序

在分析Android系统源代码的过程中，常常会对系统的一些实现原理或者执行逻辑有不确定的地方，这时候就有必要亲自写一个应用程序来验证系统的行为，从而证实系统的实现原理或者执行逻辑。在本书后面的章节中，我们也会经常结合具体的应用程序实例来分析Android系统的源代码，本节就介绍如何在Android源代码工程环境中开发Android应用程序。

本节不打算详细介绍Android应用程序的开发方法，并假设读者已经熟悉如何使用Android SDK来开发Android应用程序，具体可以参考前面在1.2小节中所介绍的*Professional Android 2 Application Development*和*Google Android SDK开发范例大全*两本书。

在Android源代码工程环境中，我们通常把实验性质的Android应用程序放在packages/experimental目录下。本书将要开发的应用程序名称为HelloAndroid，它的目录结构如下。

```
~/Android/packages/experimental/HelloAndroid
----AndroidManifest.xml
----Android.mk
----src
    ----shy/luo/hello
        ----HelloAndroid.java
----res
    ----layout
        ----main.xml
    ----values
        ----strings.xml
    ----drawable
        ----icon.png
```

Android应用程序工程一般包含有一个源代码目录src、一个资源目录res、一个配置文件AndroidManifest.xml和一个编译脚本文件Android.mk。接下来，我们就分别介绍这些文件的实现。

HelloAndroid.java

```
01 package shy.luo.hello;
02
03 import android.app.Activity;
04 import android.os.Bundle;
05 import android.util.Log;
06
07 public class HelloAndroid extends Activity {
08         private final static String LOG_TAG = "shy.luo.hello.HelloAndroid";
09
10         @Override
11         public void onCreate(Bundle savedInstanceState) {
12                 super.onCreate(savedInstanceState);
13                 setContentView(R.layout.main);
14
15                 Log.i(LOG_TAG, "HelloAndroid Activity Created.");
16         }
17 }
```

这是应用程序HelloAndroid的源代码文件，定义了一个Activity组件HelloAndroid，用作应用程序的主界面。

main.xml

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
03     android:orientation="vertical"
04     android:layout_width="fill_parent"
05     android:layout_height="fill_parent"
06     android:gravity="center">
07         <TextView
08             android:layout_width="wrap_content"
09             android:layout_height="wrap_content"
10             android:gravity="center"
11             android:text="@string/hello_android" >
12         </TextView>
13 </LinearLayout>
```

这是应用程序HelloAndroid的主界面配置文件，在屏幕中间用一个TextView控件来显示一段“Hello Android”文本。

strings.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3     <string name="app_name">HelloAndroid</string>
4     <string name="hello_android">Hello Android</string>
5 </resources>
```

这是应用程序HelloAndroid的字符串资源文件，定义了在应用程序中用到的各个字符串。

icon.png

这是应用程序HelloAndroid的图标，可以根据需要来放置不同的图片文件。

AndroidManifest.xml

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
03       package="shy.luo.hello"
04       android:versionCode="1"
05       android:versionName="1.0">
06       <application android:icon="@drawable/icon"
07             android:label="@string/app_name">
08             <activity android:name=".HelloAndroid"
09                   android:label="@string/app_name">
10                   <intent-filter>
11                         <action android:name="android.intent.action.MAIN" />
12                         <category android:name="android.intent.category.LAUNCHER" />
13                   </intent-filter>
14             </activity>
15       </application>
16 </manifest>
```

这是应用程序HelloAndroid的配置文件，应用程序HelloAndroid中所用到的Activity组件HelloAndroid需要在这里配置之后才能使用。

Android.mk

```
1 LOCAL_PATH:= $(call my-dir)
2 include $(CLEAR_VARS)
3 LOCAL_MODULE_TAGS := optional
4 LOCAL_SRC_FILES := $(call all-subdir-java-files)
5 LOCAL_PACKAGE_NAME := HelloAndroid
6 include $(BUILD_PACKAGE)
```

这是应用程序HelloAndroid的编译脚本文件，指定应用程序的名字为“HelloAndroid”。

应用程序HelloAndroid的各个文件都准备好以后，我们就可以对它们进行编译了。在Android源代码工程环境中开发Android应用程序不需要使用make命令来重新编译整个Android源代码工程，因为这将是一个漫长的过程。我们只需要单独编译新增的Android应用程序模块就可以了。在接下来的1.6小节中，我们将详细介绍如何单独编译Android源代码工程中的Android应用程序模块。

应用程序HelloAndroid经过编译，以及重新打包Android系统文件system.img之后，我们就可以使用这个新的system.img文件来启动Android模拟器了。Android模拟器启动起来之后，我们就可以在应用程序启动器中看到应用程序HelloAndroid的快捷图标了。点击应用程序HelloAndroid的快捷图标，就可以将它启动起来，如图1-2所示。

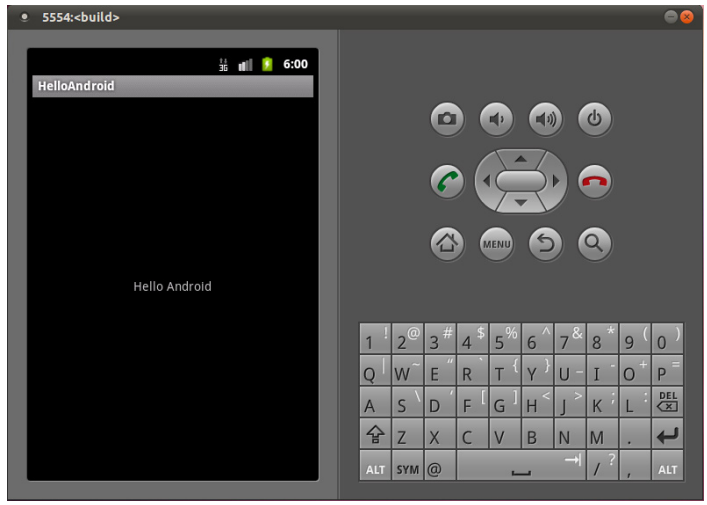


图1-2　应用程序HelloAndroid的主界面

1.6　单独编译和打包Android应用程序模块

第一次下载好Android源代码之后，我们在源代码根目录中执行make命令来对它进行编译。经过漫长的等待之后，最终就得到Android系统镜像文件system.img。如果以后修改了Android源代码工程中的某个模块或者在Android源代码工程中新增了一个模块，是否还需要执行make命令呢？答案是否定的，Android源代码工程提供了一个mmm命令来单独编译某一个模块，并提供了一个make snod命令来纯粹执行打包Android系统镜像文件system.img的操作。这样我们就可以只编译Android源代码目录中发生过修改的模块，提高编译速度。接下来，我们就介绍mmm和make snod两个命令的使用方法。

1.6.1　导入单独编译模块的mmm命令

在默认情况下，在Android源代码工程环境中，mmm命令是不可用的。我们需要在当前终端中执行Android源代码目录下的build/envsetup.sh脚本文件，然后才可以使用mmm命令。

```
USER@MACHINE:~/Android$ source  ./build/envsetup.sh
```

执行完成这个命令之后，除了获得单独编译某个模块的mmm命令之外，还有其他一些命令可以使用，如下所示。

```
- croot: Changes directory to the top of the tree.
- m: Makes from the top of the tree.
- mm: Builds all of the modules in the current directory.
- mmm: Builds all of the modules in the supplied directories.
- cgrep: Greps on all local C/C++ files.
- jgrep: Greps on all local Java files.
- resgrep: Greps on all local res/*.xml files.
- godir: Go to the directory containing a file.
```

我们可以通过指定-help选项来运行这些命令，以便可以查看它们的具体使用方法。例如，执行下面的命令就可以看到mmm命令的用法。

```
USER@MACHINE:~/Android$ mmm -help
Usage: make [options] [target] ...
Options:
  . . . . . .
```

mmm命令的用法比较丰富，但是一般情况下，我们只要在该命令后面加上要编译的模块所在的目录就可以了。

1.6.2 单独编译Android应用程序模块

在mmm命令后面加上要编译的应用程序所在的目录，就可以对该应用程序单独进行编译了，不过前提是这个目录下面必须存在一个编译脚本文件Android.mk。以前面1.5小节中的应用程序HelloAndroid为例，我们执行以下命令对它进行编译。

```
USER@MACHINE:~/Android$ mmm ./packages/experimental/HelloAndroid/
```

编译完成之后，就可以在out/target/product/generic/system/app目录下看到编译结果输出文件HelloAndroid.apk。Android系统自带的应用程序文件，即apk文件，都放在这个目录中。

如果要编译的是一个C可执行文件或者动态链接库文件，则编译结果输出文件就保存在out/target/product/generic/system/bin目录或者out/target/product/generic/system/lib目录中。此外，如果编译的动态链接库文件是一个硬件抽象层模块文件，则编译结果输出文件保存在out/target/product/generic/system/lib/hw目录中。在后面的章节中，我们将会陆续看到这些不同类型的模块的单独编译方法。

1.6.3 重新打包Android系统镜像文件

使用mmm命令单独编译好一个Android应用程序之后，还需要重新打包Android系统镜像文件system.img；否则，我们把Android模拟器启动起来后，在应用程序启动器中是看不到该Android应用程序的。

执行make snod命令，即可重新打包Android系统镜像文件system.img。

```
USER-NAME@MACHINE-NAME:~/Android$ make snod
```

重新生成的Android系统镜像文件system.img位于out/target/product/generic目录中。有了这个新的镜像文件之后，我们就可以执行emulator命令来启动Android模拟器了。Android模拟器启动完成之后，就可以在应用程序启动器中看到对应的应用程序快捷图标。继续以前面1.5小节中所介绍的应用程序HelloAndroid为例，对它进行编译，以及重新打包Android系统镜像文件system.img之后，在Android模拟器启动之后，我们就可以在应用程序启动器中看到它的快捷图标了，如图1-3所示。



图1-3 应用程序HelloAndroid的快捷图标

1.7 为真机编译ROM

前面我们针对模拟器介绍了Android源代码编译和运行的过程。为了能让自己编译的Android源代码在真机上运行，也就是常说的刷ROM，我们接下来就继续介绍为真机编译ROM的过程。

能让自己编译的Android源代码运行在真机上，是一件令人兴奋的事情。一方面，可以加深对Android系统的理解；另一方面，可以随心所欲地定制系统的行为。能够定制系统的行为，不仅能够满足特殊的手机使用需求，还能辅助日常的开发工作。

例如，我们发现自己开发的App存在Native内存泄漏，这时候就需要先定位泄漏的位置，然后再采取相应的方案进行解决。App的Native内存泄漏不同于Java内存泄漏，后者在IDE（Android Studio）中提供了完备的支持；前者则需要系统的支持才能在Android Studio中使用，也就是要将系统中的libc库替换成带内存分配追踪功能的Debug版本。具体可以参考官方文档：https://source.android.com/devices/tech/debug/native-memory。

一般来说，市面上的手机不会将Debug版本的libc库打包在系统中，因此我们就无法将它替换为正式的libc库。对于这个问题，可靠的解决方法是重新刷入一个带Debug版本libc库的ROM。这样的ROM可以通过两种途径来获得：一种是现成的，由第三方提供，例如CyanogenMod；另一种是自己从Android源代码中编译出来。

现成的符合要求的第三方ROM有时候不是那么容易找到。这时候如果我们能自力更生，从Android源代码中编译出来一个ROM，那么就可以丰衣足食了。在编译之前，可以任意修改源代码，以及改变编译选项，使得编译出来的ROM带有我们需要的功能。

在讲解为真机编译ROM之前，我们先来了解ROM的组成，如图1-4所示。

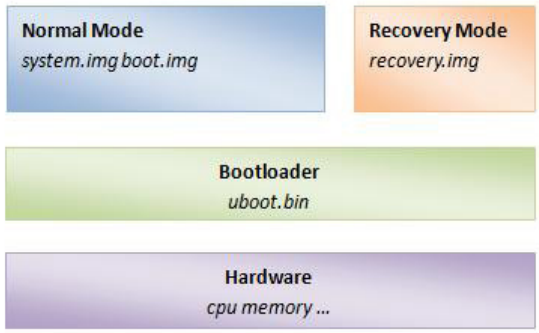


图1-4 ROM的组成

最底层的是各种硬件设备，往上一层是Bootloader。当我们按下手机电源键时，Bootloader是第一个跑起来的程序，负责初始化硬件，以及将OS启动起来。使用最广泛的Bootloader是一个叫uboot的程序，它支持非常多的体系结构。经过编译后，uboot会生成一个uboot.bin镜像，将这个镜像烧到设备上的一个特定分区中，就可以作为Bootloader使用了。

Bootloader支持交互式启动，也就是说，我们可以让Bootloader初始化完成硬件之后，不是马上去启动OS，而是停留在当前状态，等待用户输入命令告诉它接下来该干什么。这种启动方式就称为Fastboot模式。通常，在关机状态下，同时按下音量下键和电源键，可以进入Fastboot模式。

在Fastboot模式下，我们可以通过fastboot工具将一个镜像（recovery.img）刷入设备的Recovery分区。这个过程就称为刷Recovery。Recovery镜像包含一个小型的系统。通常，在关机状态下，同时按下音量上键和电源键，就可以将这个系统运行起来。这种启动方式就称为Recovery模式。

在Recovery模式下，我们可以修改正常模式下的系统，也就是平时开机进入的系统。在正常模式下，系统包含System和Boot两个核心分区。System分区包含Android运行时框架、系统App以及预装的第三方App等；而Boot分区包含内核和根文件系统。刷入System和Boot分区的两个镜像分别称为system.img和boot.img。通常将它们打包和压缩成一个zip文件，例如update.zip，并且将它上传到手机的sdcard上。这样当进入Recovery模式时，就可以在Recovery界面上使用该zip包更新在正常模式下使用的系统了。这个过程就是通常所说的刷ROM。

为手机编译一个ROM所需要的源代码可以划分为三部分，如图1–5所示。

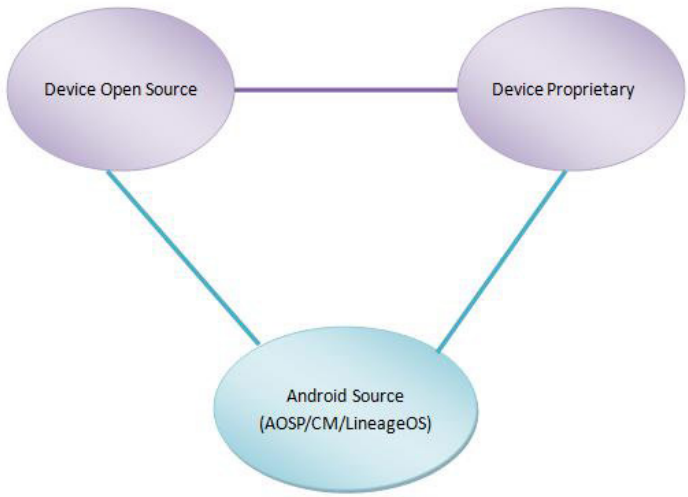


图1–5 编译ROM所需要的源代码

第一部分是Android源代码，可以使用官方AOSP的，也可以使用第三方定制的，例如CyanogenMod和LineageOS。CyanogenMod项目已经关闭了，不过CyanogenMod团队将它迁移到了新的项目LineageOS中。对普通开发者来说，LineageOS比官方AOSP更适合用来适配手机。因此，本节将使用LineageOS来为手机编译ROM。

第二部分和第三部分都是与设备相关的源代码，其中第二部分是开源的，第三部分是闭源的。开源代码大部分可以从GitHub上找到，LineageOS也包含了部分设备的开源代码。闭源代码无法获取，不过可以直接从手机上将它们对应的库文件提取出来，然后再打包到所编译的ROM里面。这些闭源代码对应的库文件称为设备私有文件。GitHub上的TheMuppets项目提供了很多设备的私有文件，我们可以直接使用。

知道了编译ROM所需要的源代码之后，我们就大概知道怎么为手机编译ROM了。LineageOS是开源项目，我们可以获得它的源代码，现在只需考虑怎么获得与设备相关的源代码即可。分两种情况考虑。第一种情况是没有与设备相关的源代码，这时候需要从零开始收集、开发它们；第二种情况是有与设备相关的源代码，这时候就可以直接编译了。

第二种情况适合初学者；第一种情况需要我们对Android系统有一定的了解之后才能做得到，同时这也是刷ROM的最高境界。本节主要介绍第二种情况，我们将会从LineageOS获得Android源代码和设备开源代码，以及从TheMuppets获得设备私有文件。这要求我们使用LineageOS和TheMuppets都支持的手机。这里选择小米3联通版，它的设备代号为cancro。

ROM编译出来后，我们需要在Recovery模式下将它刷入手机。可以使用手机自带的Recovery，也可以使用第三方Recovery。有一个著名的第三方Recovery项目TWRP，它支持大部分手机，而且支持触摸屏。其他的Recovery基本上只能通过按键（电源键和音量键）操作，不支持触摸屏。本节我们将会使用TWRP将编译出来的ROM刷入手机。

下面我们就基于LineageOS、TheMuppets和TWRP，介绍为小米3联通版编译和刷ROM的过程[1]。

1.7.1 下载LineageOS

在下载LineageOS之前，我们先按照1.3节的介绍初始化好AOSP环境。LineageOS源代码放在GitHub上，地址为：https://github.com/lineageos。

我们可以通过如下命令下载LineageOS源代码。

```
USER@MACHINE:~$ mkdir LineageOS
USER@MACHINE:~$ cd LineageOS
USER@MACHINE:~/LineageOS$ repo init -u git://github.com/LineageOS/android.git -b cm-13.0
USER@MACHINE:~/LineageOS$ repo sync
```

通过-b选项指定要下载的分支为cm-13.0，它对应的Android系统版本为6.0。从这个分支就可以看出，LineageOS确实是从CyanogenMod迁移过来的。

1.7.2 下载设备开源代码

LineageOS提供了一个breakfast命令，通过它可以下载设备开源代码。不过，在下载之前，我们需要知道手机对应的代号，这个代号可以通过网上搜索得到。对于小米3联通版来说，它的代号为cancro，可以执行如下命令下载与它相关的设备开源代码。

```
USER@MACHINE:~/LineageOS$ source build/envsetup.sh
USER@MACHINE:~/LineageOS$ breakfast cancro
```

如果LineageOS支持设备cancro，就可以将与它相关的设备源代码下载回来，并且集成在LineageOS工程中。查询LineageOS是否支持某一设备，可以在https://github.com/lineageos网页中间的搜索框中输入设备代码进行查询，如图1-6所示。

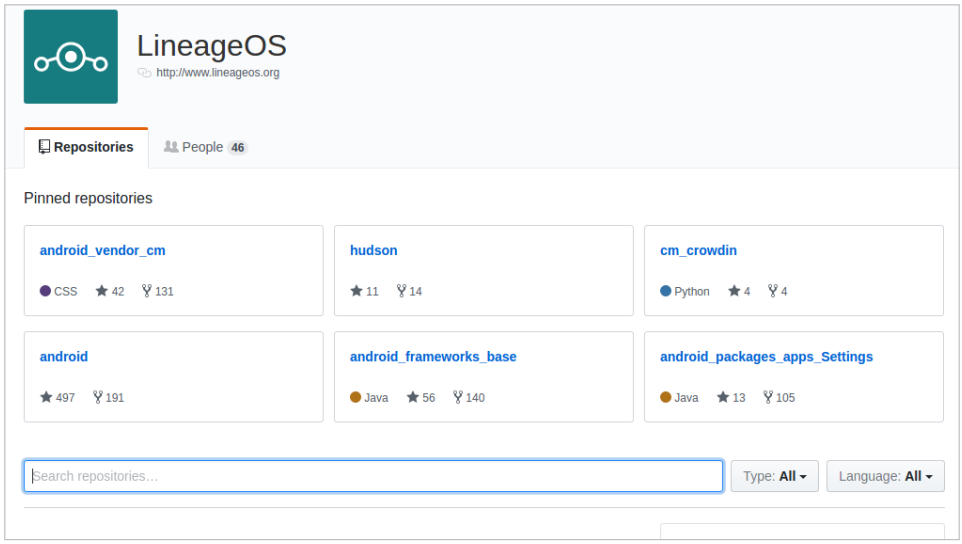


图1-6 查询LineageOS是否支持某一设备

如果支持，就可以得到查询结果。

1 刷机有风险。刷机时最好选择测试机。本节介绍的刷ROM方法，还需要手机开放Root权限。如果不是测试机，那么在刷ROM之前，先要备份好数据。

1.7.3 下载设备私有文件

打开.repo/local_manifests目录下的roomservice.xml 文件，新增如下一行。

```
<project name="TheMuppets/proprietary_vendor_xiaomi" path="vendor/xiaomi" remote="github"
revision="cm-13.0" />
```

然后执行如下命令。

```
USER@MACHINE:~/LineageOS$ repo sync
```

就可以从TheMuppets项目中下载小米设备的私有文件，并且集成在LineageOS工程中。这里的小米设备，就包含了我们要为其编译ROM的小米3联通版。

查询TheMuppets是否支持某一设备，可以在https://github.com/TheMuppets网页中间的搜索框中输入设备厂商名进行查询，如图1-7所示。

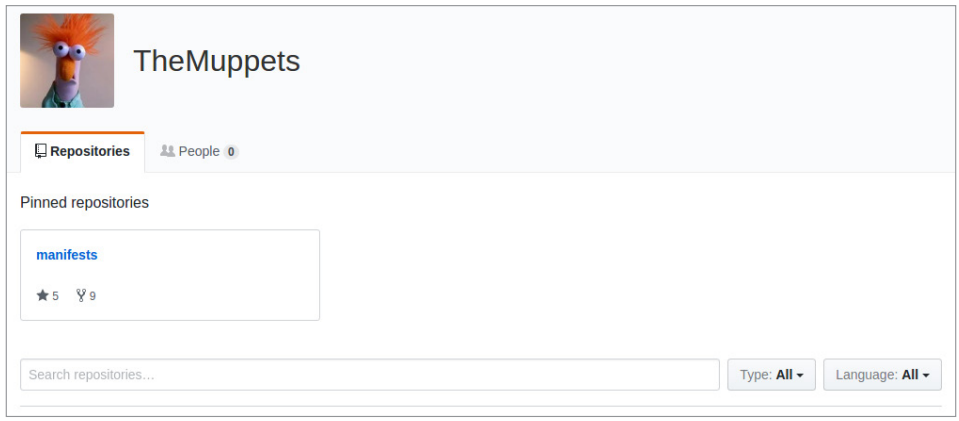


图1-7 查询TheMuppets是否支持某一设备

如果有查询结果，点击查看就可以看到是否支持相应的设备；如果没有查询结果，则说明不支持。

1.7.4 编译ROM

LineageOS提供了一个brunch命令，可以为指定的设备编译一个ROM。

```
USER@MACHINE:~/LineageOS$ source build/envsetup.sh
USER@MACHINE:~/LineageOS$ brunch cancro
```

编译完成后，在out/target/product/cancro目录下就可以得到ROM文件：lineage-13.0-<日期>-UNOFFICIAL-cancro.zip。可以通过“adb push”命令将这个ROM文件上传至手机SD卡中，以便接下来通过Recovery刷入手机。

```
USER@MACHINE:~/LineageOS$ adb push out/target/product/cancro/lineage-13.0-<日期>-UNOFFICIAL-
cancro.zip /sdcard/ lineage-13.0-<日期>-UNOFFICIAL-cancro.zip
```

1.7.5 刷入TWRP

我们通过TWRP App刷入TWRP。这时候需要授予TWRP App Root权限，这意味着系统需要开放Root功能。我们可以直接往小米3联通版手机刷一个MIUI官方的开发版ROM，下载地址为：http://www.miui.com/download-101.html。

刷入教程可以参考官方文档：http://www.miui.com/shuaji-329.html。

MIUI官方的开发版ROM携带了Root功能。在桌面上找到安全中心App，打开后进入授权管理页面即可开启Root功能。

接下来就可以安装TWRP App了，它的下载地址为：https://dl.twrp.me/twrpapp/。

TWRP App安装完成后，要再次打开安全中心App，并且进入授权管理页面，授予TWRP App Root权限。

一切准备就绪后，就可以刷入TWRP了。打开TWRP App，进入主页面，点击“TWRP FLASH”按钮，进入TWRP FLASH页面。首先通过“Select Device”按钮下载适合小米3联通版的TWRP，如图1-8所示。

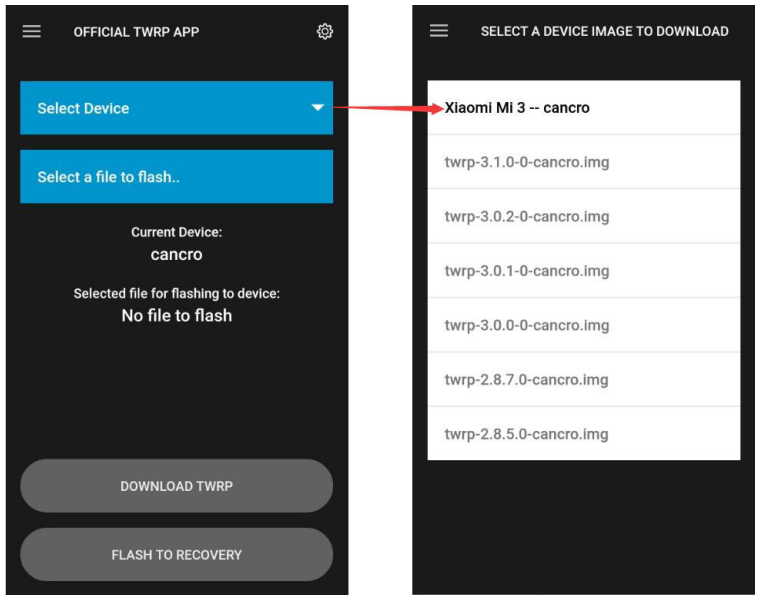


图1-8　下载TWRP

TWRP下载完成后，回到 TWRP FLASH页面，点击“Select a file to flash”按钮，选择刚才下载的TWRP文件，如图1-9所示。

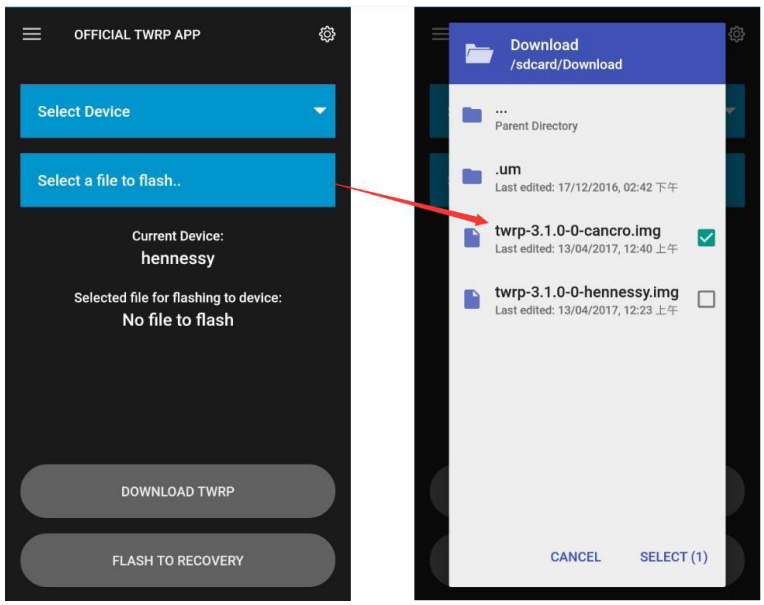


图1-9　选择TWRP文件

选择完成后，再回到 TWRP FLASH页面，点击“FLASH TO RECOVERY”按钮，就可以将TWRP刷入手机了。

1.7.6 刷入ROM

将TWRP刷入手机之后，接下来就可以将前面编译出来的ROM刷入手机了。关机后，同时按下音量上键和电源键，就可以进入到手机的Recovery模式，这时候看到的界面即为TWRP主界面，如图1–10所示。

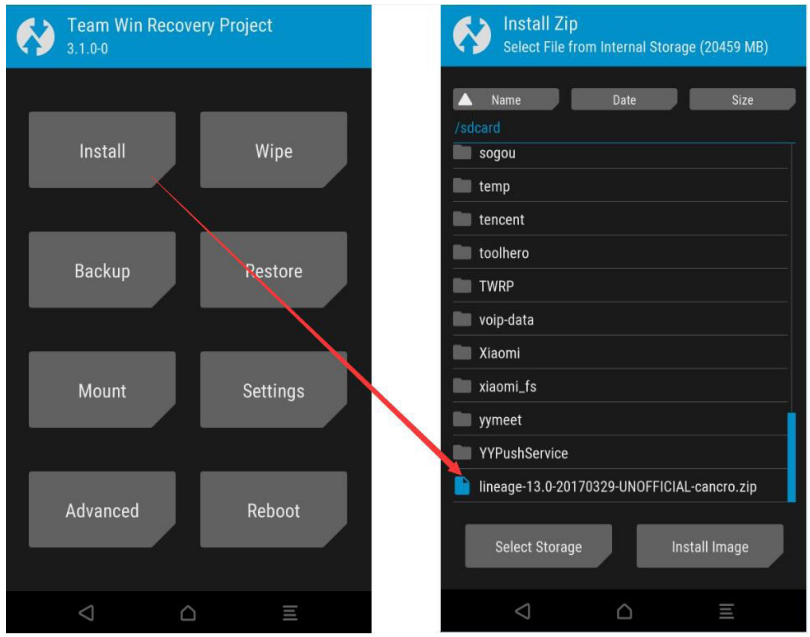


图1–10 选择ROM文件

点击“Install”按钮，在SD卡目录下找到前面上传进来的ROM文件，然后再点击“Install Image”按钮，进入另一个界面，如图1–11所示。

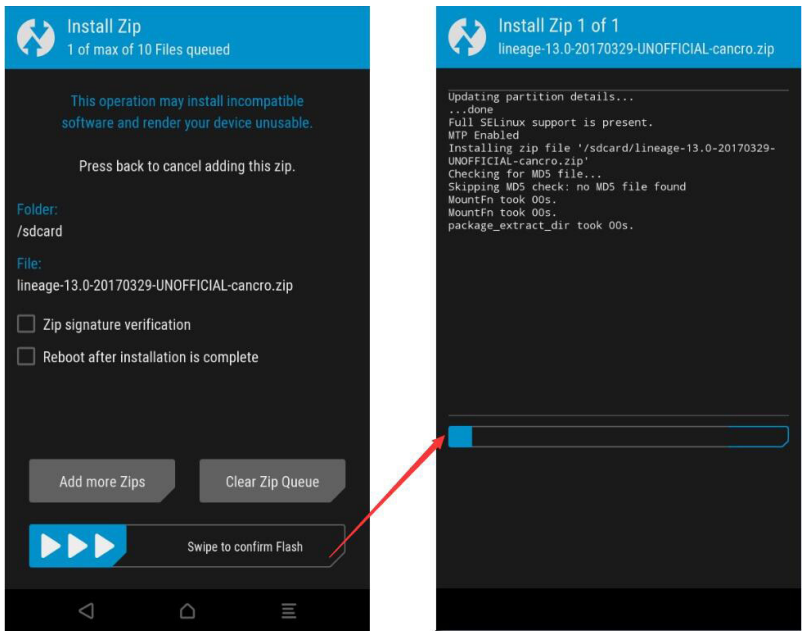


图1–11 刷入ROM文件

从左往右滑动最下面的按钮，就开始刷入ROM文件了。刷入成功后，回到TWRP主界面，即可通过“Reboot”按钮以正常模式进入系统。这个系统就是我们自己编译出来的系统。

在刷ROM过程中如果碰到错误，则可以通过以下方式解决。首先尝试清除缓存数据，如图1–12所示。

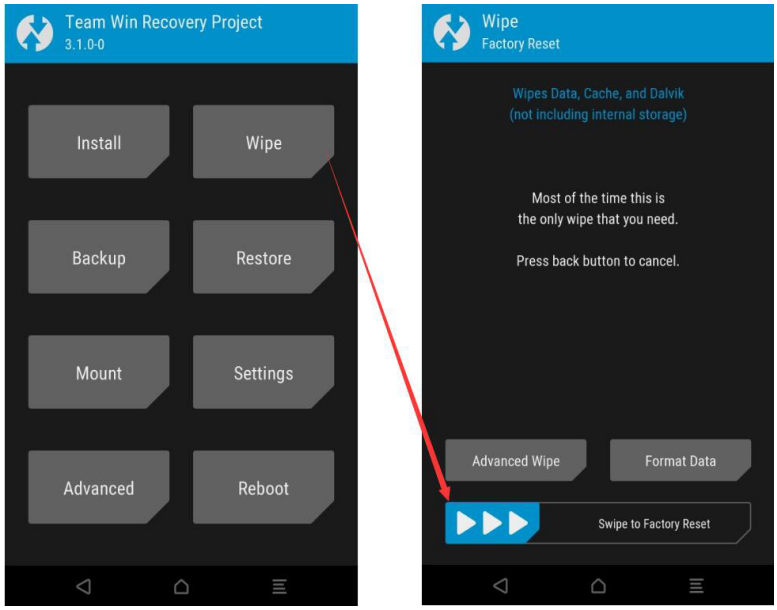


图1–12　清除缓存数据

在TWRP主界面上，点击“Wipe”按钮，进入另一个界面，在这个界面上从左往右滑动最下面的按钮，即可以清除缓存数据。缓存数据清除后，重新执行前面的刷ROM的步骤，即可完成刷ROM的过程。

如果采用上述方法仍然不能解决问题，那么就需要以正常模式进入系统，打开设置页面，找到“恢复出厂设置”选项，然后执行恢复出厂设置操作，如图1–13所示。

这一步执行完成后，再次进入Recovery模式，重新执行刷ROM操作，这时候如果编译出来的ROM没有问题，就可以成功地刷入ROM了。



图1–13　恢复出厂设置

第 2 章

硬件抽象层

Android系统的硬件抽象层（Hardware Abstract Layer，HAL）运行在用户空间中，它向下屏蔽硬件驱动模块的实现细节，向上提供硬件访问服务。通过硬件抽象层，Android系统分两层来支持硬件设备，其中一层实现在用户空间中，另一层实现在内核空间中。传统的Linux系统把对硬件的支持完全实现在内核空间中，即把对硬件的支持完全实现在硬件驱动模块中。

Android系统为什么要把对硬件的支持划分为两层来实现呢？我们知道，一方面，Linux内核源代码是遵循GPL[1]协议的，即如果我们在Android系统所使用的Linux内核中添加或者修改了代码，那么就必须将它们公开。因此，如果Android系统像其他的Linux系统一样，把对硬件的支持完全实现在硬件驱动模块中，那么就必须将这些硬件驱动模块源代码公开，这样就可能会损害移动设备厂商的利益，因为这相当于暴露了硬件的实现细节和参数。另一方面，Android系统源代码是遵循Apache License[2]协议的，它允许移动设备厂商添加或者修改Android系统源代码，而又不必公开这些代码。因此，如果把对硬件的支持完全实现在Android系统的用户空间中，那么就可以隐藏硬件的实现细节和参数。然而，这是无法做到的，因为只有内核空间才有特权操作硬件设备。一个折中的解决方案便是将对硬件的支持分别实现在内核空间和用户空间中，其中，内核空间仍然是以硬件驱动模块的形式来支持，不过它只提供简单的硬件访问通道；而用户空间以硬件抽象层模块的形式来支持，它封装了硬件的实现细节和参数。这样就可以保护移动设备厂商的利益了。

本章介绍Android系统的硬件抽象层的目的在于认识Android系统的体系结构，因为它的实现和使用依次涉及Android系统的硬件驱动模块、硬件抽象层、外部库和运行时库层、应用程序框架层和应用程序层等，如图2-1所示。

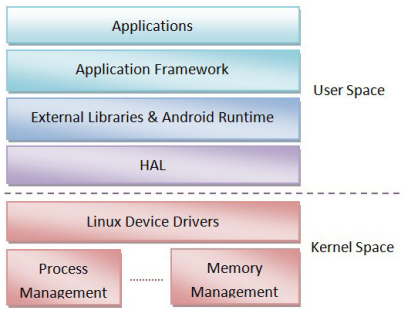


图2-1　Android系统的体系结构

1　GNU General Public License，即GNU通用公共许可证，简称为GPL，是一个被广泛使用的自由软件许可证，具体可以参考http://www.gnu.org/copyleft/gpl.html。

2　Apache Licence是一种对商业应用友好的许可，由非盈利开源组织Apache创建，具体可以参考http://www.apache.org/licenses/LICENSE-2.0.html。

接下来，我们就以硬件抽象层为中心，首先在Android系统的内核空间中为一个硬件开发驱动程序，接着在用户空间中为该硬件添加一个硬件抽象层模块，并且在应用程序框架层中添加一个硬件访问服务，最后开发一个应用程序来访问该硬件服务。这样我们就可以从下到上来认识Android系统的体系结构。

2.1 开发Android硬件驱动程序

为了方便描述，我们将为一个虚拟的字符硬件设备开发驱动程序。这个虚拟的字符硬件设备只有一个寄存器，它的大小为4字节，可读可写。由于这个字符设备是虚拟的，且只有一个寄存器，因此，我们将它称为“fake register”，并且将对应的驱动程序命名为freg。

在Android系统中开发硬件驱动程序的方法与一般Linux系统是一样的，因此，本节假设读者已经对Linux设备驱动有一定的了解，具体可以参考在前面1.1小节中介绍的*Linux Device Drivers*一书。接下来，我们首先实现驱动程序freg，然后介绍它的编译方法，最后验证它的功能正确性。

2.1.1 实现内核驱动程序模块

驱动程序freg的目录结构如下：

```
~/Android/kernel/goldfish
----drivers
     ----freg
          ----freg.h
          ----freg.c
          ----Kconfig
          ----Makefile
```

它由四个文件组成，其中freg.h和freg.c是源代码文件，Kconfig是编译选项配置文件，Makefile是编译脚本文件。下面我们就分别介绍这四个文件的实现。

freg.h

```
01 #ifndef _FAKE_REG_H_
02 #define _FAKE_REG_H_
03
04 #include <linux/cdev.h>
05 #include <linux/semaphore.h>
06
07 #define FREG_DEVICE_NODE_NAME  "freg"
08 #define FREG_DEVICE_FILE_NAME  "freg"
09 #define FREG_DEVICE_PROC_NAME  "freg"
10 #define FREG_DEVICE_CLASS_NAME "freg"
11
12 struct fake_reg_dev {
13         int val;
14         struct semaphore sem;
15         struct cdev dev;
16 };
17
18 #endif
```

这个文件定义了四个字符串常量，分别用来描述虚拟硬件设备freg在设备文件系统中的名称。此外，此文件还定义了一个结构体fake_reg_dev，用来描述虚拟硬件设备freg。在结构体fake_reg_dev中，成员变量val用来描述一个虚拟寄存器，它的类型为int；成员变量sem是一个信号量，用来同步访问虚

拟寄存器val；成员变量dev是一个标准的Linux字符设备结构体变量，用来标志该虚拟硬件设备freg的类型为字符设备。

freg.c

这个文件是驱动程序freg的实现文件，它向用户空间提供了三个接口来访问虚拟硬件设备freg中的寄存器val。第一个是proc文件系统接口，第二个是传统的设备文件系统接口，第三个是devfs文件系统接口。下面我们就分段介绍该驱动程序的实现。

文件开头包含了必要的头文件，以及定义了一些相关的变量和函数原型，它们的含义可以参考代码中的注释。

```
001 #include <linux/init.h>
002 #include <linux/module.h>
003 #include <linux/types.h>
004 #include <linux/fs.h>
005 #include <linux/proc_fs.h>
006 #include <linux/device.h>
007 #include <asm/uaccess.h>
008
009 #include "freg.h"
010
011 /*主设备号和从设备号变量*/
012 static int freg_major = 0;
013 static int freg_minor = 0;
014
015 /*设备类别和设备变量*/
016 static struct class* freg_class = NULL;
017 static struct fake_reg_dev* freg_dev = NULL;
018
019 /*传统的设备文件操作方法*/
020 static int freg_open(struct inode* inode, struct file* filp);
021 static int freg_release(struct inode* inode, struct file* filp);
022 static ssize_t freg_read(struct file* filp, char __user *buf, size_t count, loff_t* f_pos);
023 static ssize_t freg_write(struct file* filp, const char __user *buf, size_t count, loff_t* f_pos);
024
025 /*传统的设备文件操作方法表*/
026 static struct file_operations freg_fops = {
027         .owner = THIS_MODULE,
028         .open = freg_open,
029         .release = freg_release,
030         .read = freg_read,
031         .write = freg_write,
032 };
033
034 /*devfs文件系统的设备属性操作方法*/
035 static ssize_t freg_val_show(struct device* dev, struct device_attribute* attr,  char* buf);
036 static ssize_t freg_val_store(struct device* dev, struct device_attribute* attr, const
char* buf, size_t count);
037
038 /*devfs文件系统的设备属性*/
039 static DEVICE_ATTR(val, S_IRUGO | S_IWUSR, freg_val_show, freg_val_store);
```

接下来，定义用来访问虚拟硬件设备freg的传统设备文件系统接口，主要是实现freg_open、freg_release、freg_read和freg_write四个函数，其中，前面两个函数用来打开和关闭虚拟硬件设备freg，而后面两个函数用来读取和写入虚拟硬件设备freg中的寄存器val。这些函数的实现细节可以参考代码中的注释，如下所示。

```
040 /*打开设备方法*/
041 static int freg_open(struct inode* inode, struct file* filp) {
```

```
042         struct fake_reg_dev* dev;
043
044         /*将自定义设备结构体保存在文件指针的私有数据域中，以便访问设备时可以直接拿来用*/
045         dev = container_of(inode->i_cdev, struct fake_reg_dev, dev);
046         filp->private_data = dev;
047
048         return 0;
049 }
050
051 /*设备文件释放时调用，空实现*/
052 static int freg_release(struct inode* inode, struct file* filp) {
053         return 0;
054 }
055
056 /*读取设备的寄存器val的值*/
057 static ssize_t freg_read(struct file* filp, char __user *buf, size_t count, loff_t* f_pos) {
058         ssize_t err = 0;
059         struct fake_reg_dev* dev = filp->private_data;
060
061         /*同步访问*/
062         if(down_interruptible(&(dev->sem))) {
063                 return -ERESTARTSYS;
064         }
065
066         if(count < sizeof(dev->val)) {
067                 goto out;
068         }
069
070         /*将寄存器val的值复制到用户提供的缓冲区中*/
071         if(copy_to_user(buf, &(dev->val), sizeof(dev->val))) {
072                 err = -EFAULT;
073                 goto out;
074         }
075
076         err = sizeof(dev->val);
077
078 out:
079         up(&(dev->sem));
080         return err;
081 }
082
083 /*写设备的寄存器val的值*/
084 static ssize_t freg_write(struct file* filp, const char __user *buf, size_t count, loff_t* f_pos) {
085         struct fake_reg_dev* dev = filp->private_data;
086         ssize_t err = 0;
087
088         /*同步访问*/
089         if(down_interruptible(&(dev->sem))) {
090                 return -ERESTARTSYS;
091         }
092
093         if(count != sizeof(dev->val)) {
094                 goto out;
095         }
096
097         /*将用户提供的缓冲区的值写到设备寄存器中*/
098         if(copy_from_user(&(dev->val), buf, count)) {
099                 err = -EFAULT;
100                 goto out;
101         }
102
103         err = sizeof(dev->val);
104
```

```
105 out:
106         up(&(dev->sem));
107         return err;
108 }
```

接下来，我们继续定义用来访问虚拟硬件设备freg的devfs文件系统接口。这种硬件访问接口将虚拟硬件设备freg的寄存器val当作设备的一个属性，通过读写这个属性就可以达到访问设备的目的，这是通过调用freg_val_show和freg_val_store这两个函数来实现的。为了方便后面编写proc文件系统接口来访问虚拟硬件设备freg，我们定义了两个内部使用的函数__freg_get_val和__freg_set_val，它们分别用来读写虚拟硬件设备freg的寄存器val。它们的实现如下所示。

```
109 /*将寄存器val的值读取到缓冲区buf中，内部使用*/
110 static ssize_t __freg_get_val(struct fake_reg_dev* dev, char* buf) {
111         int val = 0;
112 
113         /*同步访问*/
114         if(down_interruptible(&(dev->sem))) {
115                 return -ERESTARTSYS;
116         }
117 
118         val = dev->val;
119         up(&(dev->sem));
120 
121         return snprintf(buf, PAGE_SIZE, "%d\n", val);
122 }
123 
124 /*把缓冲区buf的值写到设备寄存器val中，内部使用*/
125 static ssize_t __freg_set_val(struct fake_reg_dev* dev, const char* buf, size_t count) {
126         int val = 0;
127 
128         /*将字符串转换成数字*/
129         val = simple_strtol(buf, NULL, 10);
130 
131         /*同步访问*/
132         if(down_interruptible(&(dev->sem))) {
133                 return -ERESTARTSYS;
134         }
135 
136         dev->val = val;
137         up(&(dev->sem));
138 
139         return count;
140 }
141 
142 /*读设备属性val的值*/
143 static ssize_t freg_val_show(struct device* dev, struct device_attribute* attr, char* buf) {
144         struct fake_reg_dev* hdev = (struct fake_reg_dev*)dev_get_drvdata(dev);
145 
146         return __freg_get_val(hdev, buf);
147 }
148 
149 /*写设备属性val的值*/
150 static ssize_t freg_val_store(struct device* dev, struct device_attribute* attr, const char* buf, size_t count) {
151          struct fake_reg_dev* hdev = (struct fake_reg_dev*)dev_get_drvdata(dev);
152 
153         return __freg_set_val(hdev, buf, count);
154 }
```

接下来，我们继续定义用来访问虚拟硬件设备freg的proc文件系统接口，主要是实现freg_proc_read和freg_proc_write这两个函数。同时，我们还定义了用来在proc文件系统中创建和删除/proc/freg文件的函数freg_create_proc和freg_remove_proc。它们的实现如下所示。

```
155 /*读取设备寄存器val的值，保存到page缓冲区中*/
156 static ssize_t freg_proc_read(char* page, char** start, off_t off, int count, int* eof, void* data) {
157         if(off > 0) {
158                 *eof = 1;
159                 return 0;
160         }
161
162         return __freg_get_val(freg_dev, page);
163 }
164
165 /*把缓冲区的值buff保存到设备寄存器val中*/
166 static ssize_t freg_proc_write(struct file* filp, const char __user *buff, unsigned long len, void* data) {
167         int err = 0;
168         char* page = NULL;
169
170         if(len > PAGE_SIZE) {
171                 printk(KERN_ALERT"The buff is too large: %lu.\n", len);
172                 return -EFAULT;
173         }
174
175         page = (char*)__get_free_page(GFP_KERNEL);
176         if(!page) {
177                 printk(KERN_ALERT"Failed to alloc page.\n");
178                 return -ENOMEM;
179         }
180
181         /*先把用户提供的缓冲区的值复制到内核缓冲区中*/
182         if(copy_from_user(page, buff, len)) {
183                 printk(KERN_ALERT"Failed to copy buff from user.\n");
184                 err = -EFAULT;
185                 goto out;
186         }
187
188         err = __freg_set_val(freg_dev, page, len);
189
190 out:
191         free_page((unsigned long)page);
192         return err;
193 }
194
195 /*创建/proc/freg文件*/
196 static void freg_create_proc(void) {
197         struct proc_dir_entry* entry;
198
199         entry = create_proc_entry(FREG_DEVICE_PROC_NAME, 0, NULL);
200         if(entry) {
201                 entry->owner = THIS_MODULE;
202                 entry->read_proc = freg_proc_read;
203                 entry->write_proc = freg_proc_write;
204         }
205 }
206
207 /*删除/proc/freg文件*/
208 static void freg_remove_proc(void) {
209         remove_proc_entry(FREG_DEVICE_PROC_NAME, NULL);
210 }
```

最后，我们定义驱动程序freg的模块加载与卸载函数freg_init和freg_exit。函数freg_init主要用来注册和初始化虚拟硬件设备freg，而函数freg_exit用来反注册和释放虚拟硬件设备freg。它们的实现如下所示。

```
211 /*初始化设备*/
212 static int  __freg_setup_dev(struct fake_reg_dev* dev) {
213         int err;
214         dev_t devno = MKDEV(freg_major, freg_minor);
215 
216         memset(dev, 0, sizeof(struct fake_reg_dev));
217 
218         /*初始化字符设备*/
219         cdev_init(&(dev->dev), &freg_fops);
220         dev->dev.owner = THIS_MODULE;
221         dev->dev.ops = &freg_fops;
222 
223         /*注册字符设备*/
224         err = cdev_add(&(dev->dev),devno, 1);
225         if(err) {
226                 return err;
227         }
228 
229         /*初始化信号量和寄存器val的值*/
230         init_MUTEX(&(dev->sem));
231         dev->val = 0;
232 
233         return 0;
234 }
235 
236 /*模块加载方法*/
237 static int __init freg_init(void) {
238         int err = -1;
239         dev_t dev = 0;
240         struct device* temp = NULL;
241 
242         printk(KERN_ALERT"Initializing freg device.\n");
243 
244         /*动态分配主设备号和从设备号*/
245         err = alloc_chrdev_region(&dev, 0, 1, FREG_DEVICE_NODE_NAME);
246         if(err < 0) {
247                 printk(KERN_ALERT"Failed to alloc char dev region.\n");
248                 goto fail;
249         }
250 
251         freg_major = MAJOR(dev);
252         freg_minor = MINOR(dev);
253 
254         /*分配freg设备结构体*/
255         freg_dev = kmalloc(sizeof(struct fake_reg_dev), GFP_KERNEL);
256         if(!freg_dev) {
257                 err = -ENOMEM;
258                 printk(KERN_ALERT"Failed to alloc freg device.\n");
259                 goto unregister;
260         }
261 
262         /*初始化设备*/
263         err = __freg_setup_dev(freg_dev);
264         if(err) {
265                 printk(KERN_ALERT"Failed to setup freg device: %d.\n", err);
266                 goto cleanup;
267         }
268 
269         /*在/sys/class/目录下创建设备类别目录freg*/
270         freg_class = class_create(THIS_MODULE, FREG_DEVICE_CLASS_NAME);
271         if(IS_ERR(freg_class)) {
272                 err = PTR_ERR(freg_class);
273                 printk(KERN_ALERT"Failed to create freg device class.\n");
```

```
274                 goto destroy_cdev;
275         }
276
277         /*在/dev/目录和/sys/class/freg目录下分别创建设备文件freg*/
278         temp = device_create(freg_class, NULL, dev, NULL, "%s", FREG_DEVICE_FILE_NAME);
279         if(IS_ERR(temp)) {
280                 err = PTR_ERR(temp);
281                 printk(KERN_ALERT"Failed to create freg device.\n");
282                 goto destroy_class;
283         }
284
285         /*在/sys/class/freg/freg目录下创建属性文件val*/
286         err = device_create_file(temp, &dev_attr_val);
287         if(err < 0) {
288                 printk(KERN_ALERT"Failed to create attribute val of freg device.\n");
289                 goto destroy_device;
290         }
291
292         dev_set_drvdata(temp, freg_dev);
293
294         /*创建/proc/freg文件*/
295         freg_create_proc();
296
297         printk(KERN_ALERT"Succeeded to initialize freg device.\n");
298
299         return 0;
300
301 destroy_device:
302         device_destroy(freg_class, dev);
303 destroy_class:
304         class_destroy(freg_class);
305 destroy_cdev:
306         cdev_del(&(freg_dev->dev));
307 cleanup:
308         kfree(freg_dev);
309 unregister:
310         unregister_chrdev_region(MKDEV(freg_major, freg_minor), 1);
311 fail:
312         return err;
313 }
314
315 /*模块卸载方法*/
316 static void __exit freg_exit(void) {
317         dev_t devno = MKDEV(freg_major, freg_minor);
318
319         printk(KERN_ALERT"Destroy freg device.\n");
320
321         /*删除/proc/freg文件*/
322         freg_remove_proc();
323
324         /*销毁设备类别和设备*/
325         if(freg_class) {
326                 device_destroy(freg_class, MKDEV(freg_major, freg_minor));
327                 class_destroy(freg_class);
328         }
329
330         /*删除字符设备和释放设备内存*/
331         if(freg_dev) {
332                 cdev_del(&(freg_dev->dev));
333                 kfree(freg_dev);
334         }
335
336         /*释放设备号资源*/
```

```
337         unregister_chrdev_region(devno, 1);
338 }
339
340 MODULE_LICENSE("GPL");
341 MODULE_DESCRIPTION("Fake Register Driver");
342
343 module_init(freg_init);
344 module_exit(freg_exit);
```

驱动程序freg的源文件准备好之后，接下来就要为它编写编译选项配置文件和编译脚本文件了。

Kconfig

```
1 config FREG
2         tristate "Fake Register Driver"
3         default n
4         help
5         This is the freg driver for android system.
```

这个文件定义了驱动程序freg的编译选项。在编译驱动程序freg之前，我们可以通过执行make menuconfig命令来设置这些编译选项，以便可以指定驱动程序freg的编译方式。从这个配置文件就可以看出，驱动程序freg可以以三种方式来编译。第一种方式是直接内建在内核中；第二种方式是编译成内核模块；第三种方式是不编译到内核中。默认的编译方式为n，即不编译到内核中，因此，在编译驱动程序freg之前，我们需要执行make menuconfig命令来修改它的编译选项，以便可以将驱动程序freg内建到内核中或者以模块的方式来编译。

Makefile

```
1 obj-$(CONFIG_FREG) += freg.o
```

这是驱动程序freg的编译脚本文件，其中，$(CONFIG_FREG)是一个变量，它的值与驱动程序freg的编译选项有关。如果选择将驱动程序freg内建到内核中，那么变量$(CONFIG_FREG)的值为y；如果选择以模块的方式来编译驱动程序freg，那么变量$(CONFIG_FREG)的值为m；如果变量$(CONFIG_FREG)的值既不为y，也不为m，那么驱动程序freg就不会被编译。

至此，我们就为虚拟硬件设备freg开发了一个驱动程序。接下来，我们还需要修改内核中的Kconfig和Makefile文件来支持驱动程序freg的编译。

2.1.2　修改内核Kconfig文件

虽然我们在2.1.1小节中为驱动程序freg编写了一个Kconfig文件，但是在默认情况下，在执行make menuconfig命令配置内核编译选项时，编译系统是无法找到这个Kconfig文件的。这时候，我们需要修改内核的根Kconfig文件，使得编译系统能够找到驱动程序freg的Kconfig文件。

当执行make menuconfig命令时，编译系统会读取arch/$(ARCH)目录下的Kconfig文件，其中，$(ARCH)指向编译的目标CPU体系架构。在前面的1.4.2小节中，我们将$(ARCH)的值设置为arm，因此，就需要修改arch/arm目录下的Kconfig文件，使得编译系统可以找到驱动程序freg的Kconfig文件。打开arch/arm/Kconfig文件，找到以下两行内容：

```
menu "Device Drivers"
……
endmenu
```

在这两行内容之间添加下面一行内容，将驱动程序freg的Kconfig文件包含进来。

```
menu "Device Drivers"
source "drivers/freg/Kconfig"
……
endmenu
```

这样，当我们执行make menuconfig命令来配置内核编译选项时，编译系统就可以找到驱动程序freg中的Kconfig文件了，这时候，我们就可以配置驱动程序freg的编译方式了。

一般来说，各个CPU体系架构目录下的Kconfig文件都会通过source "drivers/Kconfig"命令把drivers目录下的Kconfig文件包含进去。例如，x86体系架构下的Kconfig文件有下面一行内容：

```
……
source "drivers/Kconfig"
……
```

这意味着当我们在内核中新增了一个驱动程序时，只需要将它的Kconfig文件包含在drivers目录下的Kconfig文件就可以了。但是arm体系架构比较特殊，它没有将drivers目录下的Kconfig文件包含到自身的Kconfig文件中，而是将drivers/Kconfig文件的内容原封不动地复制到它的Kconfig文件中。因此，考虑到兼容其他体系架构，我们最好也以同样的方式来修改drivers/Kconfig文件，即在里面添加一行：source "drivers/freg/Kconfig"。

2.1.3 修改内核Makefile文件

与Kconfig文件相似，虽然我们在2.1.1小节中为驱动程序freg编写了一个编译脚本文件Makefile，但是在默认情况下，在执行make命令编译内核时，编译系统是无法找到这个Makefile文件的。这时候，我们需要修改drivers目录下的Makefile文件，使得编译系统能够找到驱动程序freg的Makefile文件。打开drivers/Makefile文件，在里面添加以下一行内容：

```
……
obj-$(CONFIG_FREG)+= freg/
```

这样，当我们执行make命令来编译内核时，编译系统就会对驱动程序freg进行编译。

2.1.4 编译内核驱动程序模块

前面提到，在编译驱动程序freg之前，我们需要执行make menuconfig命令来配置它的编译方式。

```
USER@MACHINE:~/Android/kernel/goldfish$ make menuconfig
```

在弹出来的第一个配置界面中用上下箭头键选择“Device Drivers”项，按Enter键；接着在弹出来的第二个配置界面中继续用上下箭头键选择“Fake Register Driver”项，按Y键或者M键，就可以看到选项前面方括号中的字符变成“*”或者“M”符号，它们分别表示将驱动程序freg编译到内核中或者以模块的方式来编译。

```
| |    [*] Fake Register Driver                                    | |
……
```

注意

如果我们要以模块的方式来编译驱动程序freg，那么就必须先在第一个配置界面中选择“Enable loadable module support”选项，并且按Y键将它的值设置为true，即使得内核可以支持动态加载模块，这样才能在第二个配置界面中按M键来配置“Fake Register Driver”选项。同样，如果要使得内核支持动态卸载模块，那么就要在第一个配置界面中选择“Enable loadable module support”选项中的子选项“Module unloading”，并且按Y键将它的值设置为true。

配置完成后，保存编译配置选项，退出make menuconfig命令，然后就可以执行make命令来编译驱动程序freg了。

```
USER@MACHINE:~/Android/kernel/goldfish$ make
```

当看到下面的输出时，就说明驱动程序freg编译成功了。

```
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
```

编译得到的内核镜像文件zImage保存在arch/arm/boot目录下，我们可以使用它来启动Android模拟器。

2.1.5　验证内核驱动程序模块

编译了驱动程序freg之后，我们就可以通过proc文件系统和devfs文件系统来验证它的功能是否正确。我们首先使用前面在2.1.4小节中得到的内核镜像文件zImage来启动Android模拟器，然后用adb工具连接上它，最后就可以使用cat和echo命令来读写/proc/freg文件或者/sys/class/freg/freg/val文件的内容了，即读写虚拟硬件设备freg的寄存器val的内容。如果读出来的内容与上次写入的内容相同，就说明我们为虚拟硬件设备freg所编写的驱动程序freg是正确的。

在读写虚拟硬件设备freg的寄存器val的内容之前，我们需要检查设备上的/dev目录下是否存在一个设备文件freg。如果存在，就说明驱动程序freg成功地将虚拟硬件设备freg注册到设备文件系统中了。

```
USER@MACHINE: ~/Android$ emulator -kernel kernel/goldfish/arch/arm/boot/zImage &
USER@MACHINE: ~/Android$ adb shell
root@android:/ # cd dev
root@android:/dev # ls freg
freg
```

接下来，我们就进入到/proc目录中，首先使用cat命令读取文件freg的内容，然后使用echo命令往文件freg中写入一个新的内容，最后使用cat命令将文件freg的内容读取出来，看看是否与上次写入的内容相同。

```
root@android:/proc # cat freg
0
root@android:/proc # echo '5' > freg
root@android:/proc # cat freg
5
```

如果能看到上面的输出，就说明我们能够使用proc文件系统接口来访问虚拟硬件设备freg的寄存器val的内容，即说明前面所开发的驱动程序freg的功能是正确的。

最后，我们进入到/sys/class/freg/freg中，首先使用cat命令读取val文件的内容，然后使用echo命令往文件val中写入一个新的内容，最后使用cat命令将文件val中的内容读取出来，同样是检查它是否与上次写入的内容相同。

```
root@android:/sys/class/freg/freg # cat val
5
root@android:/sys/class/freg/freg # echo '0' > val
root@android:/sys/class/freg/freg # cat val
0
```

如果能看到上面的输出，就说明我们能够使用devfs文件系统接口来访问虚拟硬件设备freg的寄存器val的内容，同样说明前面所开发的驱动程序freg的功能是正确的。

以上两种方法只验证了驱动程序freg所提供的proc和devfs文件系统访问接口是正确的，我们还需要进一步验证它所提供的dev文件系统访问接口也是正确的，即能正常读写设备文件/dev/freg的内容。由于设备文件/dev/freg的内容是二进制格式的，因此，使用cat和echo命令来读写它的内容不够直观，在接下来的2.2小节中，我们将通过编写一个C可执行程序来直观地验证它的dev文件系统访问接口的正确性。

2.2 开发C可执行程序验证Android硬件驱动程序

在本节中，我们将通过编写一个C可执行程序来验证驱动程序freg所提供的dev文件系统接口的正确性，这是通过调用read和write函数读写设备文件/dev/freg的内容来实现的。对于Android应用程序开发者来说，可能会觉得奇怪，怎么能在Android系统中编写C语言程序呢？其实在Android源代码工程环境中，不仅可以用C/C++语言来开发可执行程序，还可以开发动态链接库，即so文件。使用adb工具命令连接上Android模拟器之后，进入到/system/bin或者/system/lib目录中，就可以看到很多可执行程序或者动态链接库文件。在接下来的2.3小节中，我们为虚拟硬件设备freg所编写的硬件抽象层模块接口其实就是一个动态链接库文件。

在Android源代码工程环境中开发的C可执行程序源文件一般保存在external目录中，因此，我们进入到external目录中，并且创建一个freg目录，用来保存我们将要开发的C可执行程序源文件。它的目录结构如下：

```
~/Android
----external
    ----freg
        ----freg.c
        ----Android.mk
```

这个C应用程序只有一个源文件 freg.c和一个编译脚本文件Android.mk。下面我们就详细分析这两个文件的内容。

freg.c

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <fcntl.h>
04
05 #define FREG_DEVICE_NAME "/dev/freg"
06
07 int main(int argc, char** argv) {
08         int fd = -1;
09         int val = 0;
10
11         fd = open(FREG_DEVICE_NAME, O_RDWR);
12         if(fd == -1) {
13                 printf("Failed to open device %s.\n", FREG_DEVICE_NAME);
14                 return -1;
15         }
16
17         printf("Read original value:\n");
18         read(fd, &val, sizeof(val));
19         printf("%d.\n\n", val);
20
21         val = 5;
22         printf("Write value %d to %s.\n\n", val, FREG_DEVICE_NAME);
23         write(fd, &val, sizeof(val));
```

```
24
25          printf("Read the value again:\n");
26          read(fd, &val, sizeof(val));
27          printf("%d.\n\n", val);
28
29          close(fd);
30
31          return 0;
32  }
```

第11行通过调用open函数，并且以读写方式打开设备文件/dev/freg；接着第18行调用read函数读取它的内容，即读取虚拟硬件设备freg的寄存器val的内容，并且将它的内容打印出来。

第23行调用write函数将一个整数5写入到虚拟硬件设备freg的寄存器val中；接着第26行和第27行再调用read和print函数将这个整数5读取并且打印出来。假设虚拟硬件设备freg的寄存器val是第一次被访问，那么如果一切正常的话，两次打印出来的内容就应该分别为0和5。

Android.mk

```
1 LOCAL_PATH := $(call my-dir)
2 include $(CLEAR_VARS)
3 LOCAL_MODULE_TAGS := optional
4 LOCAL_MODULE := freg
5 LOCAL_SRC_FILES := $(call all-subdir-c-files)
6 include $(BUILD_EXECUTABLE)
```

这是源文件freg.c的编译脚本文件，它与使用Java语言开发的Android应用程序的编译脚本的不同之处在于include命令后面所带的参数。对于C可执行程序来说，它的编译脚本中的include命令后面跟的参数的值为$(BUILD_EXECUTABLE)，表示当前要编译的是一个可执行应用程序模块，并且将编译结果保存在out/target/product/gerneric/system/bin目录中。

准备好这两个文件之后，我们就可以通过mmm和make snod命令来编译和打包这个C可执行程序了。

```
USER@MACHINE:~/Android$ mmm ./external/freg/
USER@MACHINE:~/Android$ make snod
```

编译成功后，就可以在out/target/product/gerneric/system/bin目录下看到一个freg文件；而当打包成功后，这个文件就会包含在out/target/product/gerneric目录下的Android系统镜像文件system.img中。

最后，我们就可以使用编译和打包后得到的system.img文件启动Android模拟器，然后使用adb工具连接上它，并且进入到/system/bin目录中，执行里面的freg文件来验证驱动程序freg的dev文件系统访问接口的正确性。

```
USER@MACHINE: ~/Android$ emulator -kernel kernel/goldfish/arch/arm/boot/zImage &
USER@MACHINE: ~/Android$ adb shell
root@android:/ # cd system/bin
root@android:/system/bin # ./freg
Read original value:
0.
Write value 5 to /dev/freg.
Read the value again:
5.
```

如果能够看到上面的输出，就说明驱动程序freg所提供的dev文件系统访问接口是正确的。

接下来，我们继续介绍如何为虚拟硬件设备freg编写硬件抽象层模块接口。

2.3 开发Android硬件抽象层模块

Android系统为硬件抽象层中的模块接口定义了编写规范，我们必须按照这个规范来编写自己的硬件模块接口，否则就会导致无法正常访问硬件。下面我们首先介绍硬件抽象层模块接口的编写规范，然后再按照这个规范为虚拟硬件设备freg开发硬件抽象层模块接口，并且分析硬件抽象层模块的加载过程，最后讨论硬件设备的访问权限问题。

2.3.1 硬件抽象层模块编写规范

Android系统的硬件抽象层以模块的形式来管理各个硬件访问接口。每一个硬件模块都对应有一个动态链接库文件，这些动态链接库文件的命名需要符合一定的规范。同时，在系统内部，每一个硬件抽象层模块都使用结构体hw_module_t来描述，而硬件设备则使用结构体hw_device_t来描述。接下来，我们就分别描述硬件抽象层模块文件的命名规范以及结构体hw_module_t和hw_device_t的定义。

2.3.1.1 硬件抽象层模块文件命名规范

硬件抽象层模块文件的命名规范定义在hardware/libhardware/hardware.c文件中，如下所示。

hardware/libhardware/hardware.c

```
01 /**
02  * There are a set of variant filename for modules. The form of the filename
03  * is "<MODULE_ID>.variant.so" so for the led module the Dream variants
04  * of base "ro.product.board", "ro.board.platform" and "ro.arch" would be:
05  *
06  * led.trout.so
07  * led.msm7k.so
08  * led.ARMV6.so
09  * led.default.so
10  */
11
12 static const char *variant_keys[] = {
13     "ro.hardware",  /* This goes first so that it can pick up a different
14                        file on the emulator. */
15     "ro.product.board",
16     "ro.board.platform",
17     "ro.arch"
18 };
```

这段代码和注释的意思是，硬件抽象层模块文件的命名规范为“<MODULE_ID>.variant.so”，其中，MODULE_ID表示模块的ID，variant表示四个系统属性ro.hardware、ro.product.board、ro.board.platform和ro.arch之一。系统在加载硬件抽象层模块时，依次按照ro.hardware、ro.product.board、ro.board.platform和ro.arch的顺序来取它们的属性值。如果其中的一个系统属性存在，那么就把它的值作为variant的值，然后再检查对应的文件是否存在，如果存在，那么就找到要加载的硬件抽象层模块文件了；否则，就继续查找下一个系统属性。如果这四个系统属性都不存在，或者对应于这四个系统属性的硬件抽象层模块文件都不存在，那么就使用“<MODULE_ID>.default.so”来作为要加载的硬件抽象层模块文件的名称。

系统属性ro.hardware是在系统启动时，由init进程负责设置的。它首先会读取/proc/cmdline文件，检查里面有没有一个名称为androidboot.hardware的属性，如果有，就把它的值作为属性ro.hardware的值；否则，就将/proc/cpuinfo文件的内容读取出来，并且将里面的硬件信息解析出来，即将Hardware

字段的内容作为属性ro.hardware的值。例如，在Android模拟器中，从/proc/cpuinfo文件读取出来的Hardware字段内容为goldfish，于是，init进程就会将属性ro.hardware的值设置为“goldfish”。系统属性ro.product.board、ro.board.platform和ro.arch是从/system/build.prop文件读取出来的。文件/system/build.prop是由编译系统中的编译脚本build/core/Makefile和Shell脚本build/tools/buildinfo.sh生成的，有兴趣的读者可以研究一下这两个文件，这里就不深入分析了。

2.3.1.2 硬件抽象层模块结构体定义规范

结构体hw_module_t和hw_device_t及其相关的其他结构体定义在文件hardware/libhardware/include/hardware/hardware.h中。下面我们就分别介绍这些结构体的定义。

struct hw_module_t

hardware/libhardware/include/hardware/hardware.h

```
01 /*
02  * Value for the hw_module_t.tag field
03  */
04
05 #define MAKE_TAG_CONSTANT(A,B,C,D) (((A) << 24) | ((B) << 16) | ((C) << 8) | (D))
06 #define HARDWARE_MODULE_TAG MAKE_TAG_CONSTANT('H', 'W', 'M', 'T')
07
08 /**
09  * Name of the hal_module_info
10  */
11 #define HAL_MODULE_INFO_SYM         HMI
12
13 /**
14  * Every hardware module must have a data structure named HAL_MODULE_INFO_SYM
15  * and the fields of this data structure must begin with hw_module_t
16  * followed by module specific information.
17  */
18 typedef struct hw_module_t {
19     /** tag must be initialized to HARDWARE_MODULE_TAG */
20     uint32_t tag;
21
22     /** major version number for the module */
23     uint16_t version_major;
24
25     /** minor version number of the module */
26     uint16_t version_minor;
27
28     /** Identifier of module */
29     const char *id;
30
31     /** Name of this module */
32     const char *name;
33
34     /** Author/owner/implementor of the module */
35     const char *author;
36
37     /** Modules methods */
38     struct hw_module_methods_t* methods;
39
40     /** module's dso */
41     void* dso;
42
43     /** padding to 128 bytes, reserved for future use */
44     uint32_t reserved[32-7];
45
46 } hw_module_t;
```

结构体hw_module_t中的每一个成员变量在代码中都有详细的解释，这里不再重复。不过，有五点是需要注意的。

（1）在结构体hw_module_t的定义前面有一段注释，意思是，硬件抽象层中的每一个模块都必须自定义一个硬件抽象层模块结构体，而且它的第一个成员变量的类型必须为hw_module_t。

（2）硬件抽象层中的每一个模块都必须存在一个导出符号HAL_MODULE_IFNO_SYM，即“HMI”，它指向一个自定义的硬件抽象层模块结构体。后面我们在分析硬件抽象层模块的加载过程时，将会看到这个导出符号的意义。

（3）结构体hw_module_t的成员变量tag的值必须设置为HARDWARE_MODULE_TAG，即设置为一个常量值（'H' <<24 | 'W'<<16 | 'M'<<8 | 'T'），用来标志这是一个硬件抽象层模块结构体。

（4）结构体hw_module_t的成员变量dso用来保存加载硬件抽象层模块后得到的句柄值。前面提到，每一个硬件抽象层模块都对应有一个动态链接库文件。加载硬件抽象层模块的过程实际上就是调用dlopen函数来加载与其对应的动态链接库文件的过程。在调用dlclose函数来卸载这个硬件抽象层模块时，要用到这个句柄值，因此，我们在加载时需要将它保存起来。

（5）结构体hw_module_t的成员变量methods定义了一个硬件抽象层模块的操作方法列表，它的类型为hw_module_methods_t，接下来我们就介绍它的定义。

struct hw_module_methods_t

hardware/libhardware/include/hardware/hardware.h

```
1 typedef struct hw_module_methods_t {
2     /** Open a specific device */
3     int (*open)(const struct hw_module_t* module, const char* id,
4             struct hw_device_t** device);
5
6 } hw_module_methods_t;
```

结构体hw_module_methods_t只有一个成员变量，它是一个函数指针，用来打开硬件抽象层模块中的硬件设备。其中，参数module表示要打开的硬件设备所在的模块；参数id表示要打开的硬件设备的ID；参数device是一个输出参数，用来描述一个已经打开的硬件设备。由于一个硬件抽象层模块可能会包含多个硬件设备，因此，在调用结构体hw_module_methods_t的成员变量open来打开一个硬件设备时，我们需要指定它的ID。硬件抽象层中的硬件设备使用结构体hw_device_t来描述，接下来我们就介绍它的定义。

struct hw_device_t

hardware/libhardware/include/hardware/hardware.h

```
01 #define HARDWARE_DEVICE_TAG MAKE_TAG_CONSTANT('H', 'W', 'D', 'T')
02
03 /**
04  * Every device data structure must begin with hw_device_t
05  * followed by module specific public methods and attributes.
06  */
07 typedef struct hw_device_t {
08     /** tag must be initialized to HARDWARE_DEVICE_TAG */
09     uint32_t tag;
10
11     /** version number for hw_device_t */
12     uint32_t version;
13
14     /** reference to the module this device belongs to */
15     struct hw_module_t* module;
16
17     /** padding reserved for future use */
```

```
18      uint32_t reserved[12];
19
20      /** Close this device */
21      int (*close)(struct hw_device_t* device);
22
23  } hw_device_t;
```

结构体hw_device_t中的每一个成员变量在代码中都有详细的解释，这里不再重复。不过，有三点是需要注意的。

（1）硬件抽象层模块中的每一个硬件设备都必须自定义一个硬件设备结构体，而且它的第一个成员变量的类型必须为hw_device_t。

（2）结构体hw_device_t的成员变量tag的值必须设置为HARDWARE_DEVICE_TAG，即设置为一个常量值（'H' <<24 | 'W'<<16 | 'D'<<8 | 'T'），用来标志这是一个硬件抽象层中的硬件设备结构体。

（3）结构体hw_device_t的成员变量close是一个函数指针，它用来关闭一个硬件设备。

注意

硬件抽象层中的硬件设备是由其所在的模块提供接口来打开的，而关闭则是由硬件设备自身提供接口来完成的。

至此，硬件抽象层模块接口的编写规范就介绍完了。接下来，我们就可以为虚拟硬件设备freg编写硬件抽象层模块接口了。

2.3.2　编写硬件抽象层模块接口

每一个硬件抽象层模块在内核中都对应有一个驱动程序，硬件抽象层模块就是通过这些驱动程序来访问硬件设备的，它们是通过读写设备文件来进行通信的。

硬件抽象层中的模块接口源文件一般保存在hardware/libhardware目录中。为了方便起见，我们将虚拟硬件设备freg在硬件抽象层中的模块名称定义为freg，它的目录结构如下：

```
~/Android/hardware/libhardware
----include
     ----hardware
          ----freg.h
Modules
     ----freg
          ----freg.cpp
          ----Android.mk
```

它由三个文件组成，其中，freg.h和freg.cpp是源代码文件，而Android.mk是模块的编译脚本文件。下面我们就分别介绍这三个文件的内容。

freg.h

```
01 #ifndef ANDROID_FREG_INTERFACE_H
02 #define ANDROID_FREG_INTERFACE_H
03
04 #include <hardware/hardware.h>
05
06 __BEGIN_DECLS
07
08 /*定义模块ID*/
09 #define FREG_HARDWARE_MODULE_ID "freg"
10
11 /*定义设备ID*/
12 #define FREG_HARDWARE_DEVICE_ID "freg"
```

```
13
14 /*自定义模块结构体*/
15 struct freg_module_t {
16         struct hw_module_t common;
17 };
18
19 /*自定义设备结构体*/
20 struct freg_device_t {
21         struct hw_device_t common;
22         int fd;
23         int (*set_val)(struct freg_device_t* dev, int val);
24         int (*get_val)(struct freg_device_t* dev, int* val);
25 };
26
27 __END_DECLS
28
29 #endif
```

这个文件中的常量和结构体都是按照硬件抽象层模块编写规范来定义的。宏FREG_HARDWARE_MODULE_ID和FREG_HARDWARE_DEVICE_ID分别用来描述模块ID和设备ID。结构体freg_module_t用来描述自定义的模块结构体，它的第一个成员变量的类型为hw_module_t。结构体freg_device_t用来描述虚拟硬件设备freg，它的第一个成员变量的类型为hw_device_t。此外，结构体freg_device_t还定义了其他三个成员变量，其中，成员变量fd是一个文件描述符，用来描述打开的设备文件/dev/freg，成员变量set_val和get_val是函数指针，它们分别用来写和读虚拟硬件设备freg的寄存器val的内容。

freg.cpp

这是硬件抽象层模块freg的实现文件，我们分段来阅读。

文件首先包含相关头文件并且定义相关结构体变量。

```
01 #define LOG_TAG "FregHALStub"
02
03 #include <hardware/hardware.h>
04 #include <hardware/freg.h>
05
06 #include <fcntl.h>
07 #include <errno.h>
08
09 #include <cutils/log.h>
10 #include <cutils/atomic.h>
11
12 #define DEVICE_NAME "/dev/freg"
13 #define MODULE_NAME "Freg"
14 #define MODULE_AUTHOR "shyluo@gmail.com"
15
16 /*设备打开和关闭接口*/
17 static int freg_device_open(const struct hw_module_t* module, const char* id, struct hw_device_t** device);
18 static int freg_device_close(struct hw_device_t* device);
19
20 /*设备寄存器读写接口*/
21 static int freg_get_val(struct freg_device_t* dev, int* val);
22 static int freg_set_val(struct freg_device_t* dev, int val);
23
24 /*定义模块操作方法结构体变量*/
25 static struct hw_module_methods_t freg_module_methods = {
26         open: freg_device_open
27 };
28
29 /*定义模块结构体变量*/
```

```
30 struct freg_module_t HAL_MODULE_INFO_SYM = {
31         common: {
32                 tag: HARDWARE_MODULE_TAG,
33                 version_major: 1,
34                 version_minor: 0,
35                 id: FREG_HARDWARE_MODULE_ID,
36                 name: MODULE_NAME,
37                 author: MODULE_AUTHOR,
38                 methods: &freg_module_methods,
39         }
40 };
```

在这段代码中，最值得关注的就是模块变量HAL_MODULE_INFO_SYM的定义。按照硬件抽象层模块编写规范，每一个硬件抽象层模块必须导出一个名称为HAL_MODULE_INFO_SYM的符号，它指向一个自定义的硬件抽象层模块结构体，而且它的第一个类型为hw_module_t的成员变量的tag值必须设置为HARDWARE_MODULE_TAG。除此之外，还初始化了这个硬件抽象层模块结构体的版本号、ID、名称、作者和操作方法列表等。

虚拟硬件设备freg的打开和关闭分别由函数freg_device_open和freg_device_close来实现，如下所示。

```
41 static int freg_device_open(const struct hw_module_t* module, const char* id, struct hw_device_t** device) {
42         if(!strcmp(id, FREG_HARDWARE_DEVICE_ID)) {
43                 struct freg_device_t* dev;
44
45                 dev = (struct freg_device_t*)malloc(sizeof(struct freg_device_t));
46                 if(!dev) {
47                         LOGE("Failed to alloc space for freg_device_t.");
48                         return -EFAULT;
49                 }
50
51                 memset(dev, 0, sizeof(struct freg_device_t));
52
53                 dev->common.tag = HARDWARE_DEVICE_TAG;
54                 dev->common.version = 0;
55                 dev->common.module = (hw_module_t*)module;
56                 dev->common.close = freg_device_close;
57                 dev->set_val = freg_set_val;
58                 dev->get_val = freg_get_val;
59
60                 if((dev->fd = open(DEVICE_NAME, O_RDWR)) == -1) {
61                         LOGE("Failed to open device file /dev/freg -- %s.", strerror(errno));
62                         free(dev);
63                         return -EFAULT;
64                 }
65
66                 *device = &(dev->common);
67
68                 LOGI("Open device file /dev/freg successfully.");
69
70                 return 0;
71         }
72
73         return -EFAULT;
74 }
75
76 static int freg_device_close(struct hw_device_t* device) {
77         struct freg_device_t* freg_device = (struct freg_device_t*)device;
78         if(freg_device) {
79                 close(freg_device->fd);
80                 free(freg_device);
81         }
82
83         return 0;
84 }
```

前面提到，一个硬件抽象层模块可能会包含多个硬件设备，而这些硬件设备的打开操作都是由函数freg_device_open来完成的，因此，函数freg_device_open会根据传进来的参数id来判断要打开哪一个硬件设备。

在硬件抽象层模块freg中，只有一个虚拟硬件设备freg，它使用结构体freg_device_t来描述。因此，函数freg_device_open发现参数id与虚拟硬件设备freg的ID值匹配以后，就会分配一个freg_device_t结构体，并且对它的成员变量进行初始化。按照硬件抽象层模块编写规范，硬件抽象层中的硬件设备标签（dev->common.tag）必须设置为HARDWARE_DEVICE_TAG。除此之外，我们还将虚拟硬件设备freg的关闭函数设置为freg_device_close，并且将它的读写函数设置为freg_get_val和freg_set_val。

初始化完成用来描述虚拟硬件设备freg的结构体freg_device_t之后，我们就可以调用open函数来打开虚拟硬件设备文件/dev/freg了，并且将得到的文件描述符保存在结构体freg_device_t的成员变量fd中。

虚拟硬件设备freg的关闭函数freg_device_close的实现比较简单，它主要是关闭设备文件/dev/freg，以及释放设备在打开时所分配的资源。

虚拟硬件设备freg的读写函数freg_get_val和freg_set_val的实现如下所示。

```
85 static int freg_get_val(struct freg_device_t* dev, int* val) {
86         if(!dev) {
87                 LOGE("Null dev pointer.");
88                 return -EFAULT;
89         }
90
91         if(!val) {
92                 LOGE("Null val pointer.");
93                 return -EFAULT;
94         }
95
96         read(dev->fd, val, sizeof(*val));
97
98         LOGI("Get value %d from device file /dev/freg.", *val);
99
100         return 0;
101 }
102
103 static int freg_set_val(struct freg_device_t* dev, int val) {
104         if(!dev) {
105                 LOGE("Null dev pointer.");
106                 return -EFAULT;
107         }
108
109         LOGI("Set value %d to device file /dev/freg.", val);
110         write(dev->fd, &val, sizeof(val));
111
112         return 0;
113 }
```

这两个函数分别通过调用read和write函数来实现读写虚拟硬件设备freg的寄存器val的内容。

Android.mk

```
1 LOCAL_PATH := $(call my-dir)
2 include $(CLEAR_VARS)
3 LOCAL_MODULE_TAGS := optional
4 LOCAL_PRELINK_MODULE := false
5 LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)/hw
6 LOCAL_SHARED_LIBRARIES := liblog
7 LOCAL_SRC_FILES := freg.cpp
8 LOCAL_MODULE := freg.default
9 include $(BUILD_SHARED_LIBRARY)
```

这是硬件抽象层模块freg的编译脚本文件。第9行指定include命令的参数为$(BUILD_SHARED_LIBRARY)，表示要将该硬件抽象层模块编译成一个动态链接库文件，名称为freg.default，并且保存在$(TARGET_OUT_SHARED_LIBRARIES)/hw目录下，即out/target/product/generic/system/lib/hw目录下。

注意

我们将硬件抽象层模块freg对应的文件名称定义为freg.default，编译成功后，系统就会自动在后面加后缀.so，于是就得到了一个freg.default.so文件。根据硬件抽象层模块文件的命名规范，当我们要加载硬件抽象层模块freg时，只需要指定它的ID值，即“freg”，系统就会根据一定的规则成功地找到要加载的freg.default.so文件。

硬件抽象层模块freg的所有文件都准备好之后，我们就可以执行mmm和make snod命令对它进行编译和打包了。

```
USER@MACHINE:~/Android$ mmm ./hardware/libhardware/modules/freg/
USER@MACHINE:~/Android$ make snod
```

最终就可以在out/target/product/generic/system/lib/hw目录下得到一个freg.default.so文件。

2.3.3　硬件抽象层模块的加载过程

学习Android硬件抽象层模块的加载过程有助于理解它的编写规范以及实现原理。Android系统中的硬件抽象层模块是由系统统一加载的，当调用者需要加载这些模块时，只要指定它们的ID值就可以了。

在Android硬件抽象层中，负责加载硬件抽象层模块的函数是hw_get_module，它的原型如下：

hardware/libhardware/include/hardware/hardware.h

```
1 /**
2  * Get the module info associated with a module by id.
3  * @return: 0 == success, <0 == error and *pHmi == NULL
4  */
5 int hw_get_module(const char *id, const struct hw_module_t **module);
```

它有id和module两个参数。其中，id是输入参数，表示要加载的硬件抽象层模块ID；module是输出参数，如果加载成功，那么它指向一个自定义的硬件抽象层模块结构体。函数的返回值是一个整数，如果等于0，则表示加载成功；如果小于0，则表示加载失败。

下面我们就开始分析hw_get_module函数的实现。

hardware/libhardware/hardware.c

```
01 /** Base path of the hal modules */
02 #define HAL_LIBRARY_PATH1 "/system/lib/hw"
03 #define HAL_LIBRARY_PATH2 "/vendor/lib/hw"
04
05 static const char *variant_keys[] = {
06     "ro.hardware",  /* This goes first so that it can pick up a different
07                        file on the emulator. */
08     "ro.product.board",
09     "ro.board.platform",
10     "ro.arch"
11 };
12
13 static const int HAL_VARIANT_KEYS_COUNT =
14     (sizeof(variant_keys)/sizeof(variant_keys[0]));
15
16 int hw_get_module(const char *id, const struct hw_module_t **module)
17 {
18     int status;
19     int i;
20     const struct hw_module_t *hmi = NULL;
21     char prop[PATH_MAX];
```

```
22      char path[PATH_MAX];
23
24      /*
25       * Here we rely on the fact that calling dlopen multiple times on
26       * the same .so will simply increment a refcount (and not load
27       * a new copy of the library).
28       * We also assume that dlopen() is thread-safe.
29       */
30
31      /* Loop through the configuration variants looking for a module */
32      for (i=0 ; i<HAL_VARIANT_KEYS_COUNT+1 ; i++) {
33          if (i < HAL_VARIANT_KEYS_COUNT) {
34              if (property_get(variant_keys[i], prop, NULL) == 0) {
35                  continue;
36              }
37
38              snprintf(path, sizeof(path), "%s/%s.%s.so",
39                      HAL_LIBRARY_PATH1, id, prop);
40              if (access(path, R_OK) == 0) break;
41
42              snprintf(path, sizeof(path), "%s/%s.%s.so",
43                       HAL_LIBRARY_PATH2, id, prop);
44              if (access(path, R_OK) == 0) break;
45          } else {
46              snprintf(path, sizeof(path), "%s/%s.default.so",
47                       HAL_LIBRARY_PATH1, id);
48              if (access(path, R_OK) == 0) break;
49          }
50      }
51
52      status = -ENOENT;
53      if (i < HAL_VARIANT_KEYS_COUNT+1) {
54          /* load the module, if this fails, we're doomed, and we should not try
55           * to load a different variant. */
56          status = load(id, path, module);
57      }
58
59      return status;
60  }
```

在前面的2.3.1.1小节中，我们已经见过数组variant_keys的定义了，它用来组装要加载的硬件抽象层模块的文件名称。常量HAL_VARIANT_KEYS_COUNT表示数组variant_keys的大小。

宏HAL_LIBRARY_PATH1和HAL_LIBRARY_PATH2用来定义要加载的硬件抽象层模块文件所在的目录。在前面的2.3.2小节中提到，编译好的模块文件位于out/target/product/generic/system/lib/hw目录中，而这个目录经过打包后，就对应于设备上的/system/lib/hw目录。宏HAL_LIBRARY_PATH2所定义的目录为/vendor/lib/hw，用来保存设备厂商所提供的硬件抽象层模块接口文件。

函数第32行到第50行的for循环根据数组variant_keys在HAL_LIBRARY_PATH1和HAL_LIBRARY_PATH2目录中检查对应的硬件抽象层模块文件是否存在，如果存在，则结束for循环；第56行调用load函数来执行加载硬件抽象层模块的操作。

我们以在Android模块器中加载硬件抽象层模块freg为例，来分析硬件抽象层模块的加载过程。在调用hw_get_module函数加载硬件抽象层模块freg时，传入的参数id的值为FREG_HARDWARE_MODULE_ID，即“freg”。在上面的for循环中，首先找到通过property_get函数获得的系统属性“ro.hardware”的值。在Android模拟器中，这个属性的值定义为“goldfish”，于是通过第38和39两行的snprintf函数，就得到变量path的值为“/system/lib/hw/freg.goldfish.so”。第40行调用access函数判断文件/system/lib/hw/freg.goldfish.so是否存在，如果存在，就跳出循环；否则，再通过第42行到第44行的代码来判断文件/vendor/lib/hw/freg.goldfish.so是否存在，如果存在，那么也会跳出循环，因为要加载的硬件抽象层模块

文件已经找到了。如果这两个文件都不存在，那么按照相同的方法来依次查找数组variant_keys中其他元素所对应的硬件抽象层模块文件是否存在。如果数组variant_keys中的所有元素对应的硬件抽象层模块文件都不存在，那么第46行到第48行的代码就会在/system/lib/hw目录中检查是否存在一个freg.default.so文件。如果也不存在，那么硬件抽象层模块freg的加载就失败了。

找到了硬件抽象层模块文件之后，第56行就调用load函数来执行硬件抽象层模块的加载操作，它的实现如下所示。

hardware/libhardware/hardware.c

```
01 static int load(const char *id,
02         const char *path,
03         const struct hw_module_t **pHmi)
04 {
05     int status;
06     void *handle;
07     struct hw_module_t *hmi;
08
09     /*
10      * load the symbols resolving undefined symbols before
11      * dlopen returns. Since RTLD_GLOBAL is not or'd in with
12      * RTLD_NOW the external symbols will not be global
13      */
14     handle = dlopen(path, RTLD_NOW);
15     if (handle == NULL) {
16         char const *err_str = dlerror();
17         LOGE("load: module=%s\n%s", path, err_str?err_str:"unknown");
18         status = -EINVAL;
19         goto done;
20     }
21
22     /* Get the address of the struct hal_module_info. */
23     const char *sym = HAL_MODULE_INFO_SYM_AS_STR;
24     hmi = (struct hw_module_t *)dlsym(handle, sym);
25     if (hmi == NULL) {
26         LOGE("load: couldn't find symbol %s", sym);
27         status = -EINVAL;
28         goto done;
29     }
30
31     /* Check that the id matches */
32     if (strcmp(id, hmi->id) != 0) {
33         LOGE("load: id=%s != hmi->id=%s", id, hmi->id);
34         status = -EINVAL;
35         goto done;
36     }
37
38     hmi->dso = handle;
39
40     /* success */
41     status = 0;
42
43     done:
44     if (status != 0) {
45         hmi = NULL;
46         if (handle != NULL) {
47             dlclose(handle);
48             handle = NULL;
49         }
50     } else {
51         LOGV("loaded HAL id=%s path=%s hmi=%p handle=%p",
52                 id, path, *pHmi, handle);
53     }
54
```

```
55      *pHmi = hmi;
56
57      return status;
58  }
```

前面提到，硬件抽象层模块文件实际上是一个动态链接库文件，即so文件。因此，第14行调用dlopen函数将它加载到内存中。加载完成这个动态链接库文件之后，第24行就调用dlsym函数来获得里面名称为HAL_MODULE_INFO_SYM_AS_STR的符号。这个HAL_MODULE_INFO_SYM_AS_STR符号指向的是一个自定义的硬件抽象层模块结构体，它包含了对应的硬件抽象层模块的所有信息。HAL_MODULE_INFO_SYM_AS_STR是一个宏，它的值定义为“HMI”。根据硬件抽象层模块的编写规范，每一个硬件抽象层模块都必须包含一个名称为“HMI”的符号，而且这个符号的第一个成员变量的类型必须定义为hw_module_t，因此，第24行可以安全地将模块中的HMI符号转换为一个hw_module_t结构体指针。

获得了这个hw_module_t结构体指针之后，第32行调用strcmp函数来验证加载得到的硬件抽象层模块ID是否与所要求加载的硬件抽象层模块ID一致。如果不一致，就说明出错了，函数返回一个错误值-EINVAL。最后，第38行将成功加载后得到的模块句柄值handle保存在hw_module_t结构体指针hmi的成员变量dso中，然后将它返回给调用者。

至此，硬件抽象层模块的加载过程就介绍完了。接下来，我们分析在硬件抽象层模块的加载过程中，经常会碰到的一个权限问题，即无法调用open函数打开对应的设备文件。

2.3.4 处理硬件设备访问权限问题

在硬件抽象层模块中，我们是调用open函数来打开对应的设备文件的。例如，在2.3.2小节中开发的硬件抽象层模块freg中，函数freg_device_open调用open函数来打开设备文件/dev/freg。

```
60 if((dev->fd = open(DEVICE_NAME, O_RDWR)) == -1) {
61       LOGE("Failed to open device file /dev/freg -- %s.", strerror(errno));
62       free(dev);
63       return -EFAULT;
64 }
```

如果不修改设备文件/dev/freg的访问权限，那么应用程序调用freg_device_open函数打开设备文件/def/freg就会失败，从第61行的日志输出可以看到下面的内容：

```
Failed to open /dev/freg -- Permission denied.
```

这表示当前用户没有权限打开设备文件/dev/freg。在默认情况下，只有root用户才有权限访问系统的设备文件。由于一般的应用程序是没有root用户权限的，因此，这里就会提示没有权限打开设备文件/dev/freg。

解决这个问题的办法是，赋予root之外的其他用户访问设备文件/dev/freg的权限。我们知道，在Linux系统中，可以通过udev规则在系统启动时修改设备文件的访问权限[1]。然而，Android系统并没有实现udev机制，因此，我们就不可以通过定义udev规则来赋予root之外的其他用户访问设备文件/dev/freg的权限。不过，Android提供了另外的一个uevent机制，可以在系统启动时修改设备文件的访问权限。

在system/core/rootdir目录下有一个名为ueventd.rc的配置文件，我们可以在里面增加以下一行内容

1 udev是Linux 2.6内核新增的一个功能，用来替代原来的devfs，是Linux系统默认的设备管理工具。udev 机制以守护进程的形式运行，通过侦听内核发出来的uevent来管理/dev目录下的设备文件，包括添加或者删除设备文件、修改设备文件的访问权限等。

来修改设备文件/dev/freg的访问权限。

```
/dev/freg                 0666   root       root
```

这表示所有的用户均可以访问设备文件/dev/freg，即可以打开设备文件/dev/freg，以及读写它的内容。这样，除了root用户之外，系统中的其他用户也可以调用freg_device_open函数来打开设备文件/dev/freg。

修改了ueventd.rc文件后，需要重新编译Android源代码工程，这样新修改的设备文件/dev/freg的访问权限才能生效。这里，我们介绍一种不必重新编译Android源代码工程就可以使得修改后的设备文件/dev/freg的访问权限生效的方法。

在编译Android源代码工程时，文件system/core/rootdir/ueventd.rc会被复制到out/target/product/generic/root目录下，并且最终打包在ramdisk.img镜像文件中。当Android系统启动时，会把ramdisk.img镜像文件中的ueventd.rc文件安装在设备根目录中，并且由init进程来解析它的内容和修改相应的设备文件的访问权限。因此，只要我们能够修改ramdisk.img镜像文件中ueventd.rc文件的内容，就可以修改设备文件/dev/freg的访问权限。接下来就详细介绍修改ramdisk.img镜像文件中ueventd.rc文件的方法。

1. 解压ramdisk.img镜像文件

镜像文件ramdisk.img是一个gzip文件，因此，我们可以执行gunzip命令对它进行解压。

```
USER@MACHINE:~/Android$ mv ./out/target/product/generic/ramdisk.img ./ramdisk.img.gz
USER@MACHINE:~/Android$ gunzip ./ramdisk.img.gz
```

我们先将ramdisk.img改名为ramdisk.img.gz，然后调用gunzip命令对它进行解压。解压后得到的ramdisk.img文件保存在~/Android目录中。

2. 还原ramdisk.img镜像文件

解压后得到的ramdisk.img文件是一个cpio[1]格式的归档文件，因此，我们可以执行cpio命令对它解除归档。

```
USER@MACHINE:~/Android $ mkdir ramdisk
USER@MACHINE:~/Android $ cd ./ramdisk/
USER@MACHINE:~/Android/ramdisk$ cpio -i -F ../ramdisk.img
```

解除归档后得到的文件保存在~/Android/ramdisk目录中。

3. 修改ueventd.rc文件

进入到~/Android/ramdisk目录中，找到ueventd.rc文件，并且往里面增加以下一行内容：

```
/dev/freg                 0666   root       root
```

这一行内容赋予了系统中的所有用户访问设备文件/dev/freg的权限。

4. 重新打包ramdisk.img镜像文件

重新打包ramdisk.img镜像文件的过程其实就是第1步和第2步的逆过程，即先把ramdisk目录归档成cpio文件，然后压缩成gzip文件。

```
USER@MACHINE:~/Android/ramdisk$ rm -f ../ramdisk.img
USER@MACHINE:~/Android/ramdisk$ find . | cpio -o -H newc > ../ramdisk.img.unzip
USER@MACHINE:~/Android/ramdisk$ cd ..
USER@MACHINE:~/Android$ gzip -c ./ramdisk.img.unzip > ./ramdisk.img.gz
```

1　cpio是一种包含其他文件和有关信息的归档文件，具体可以参考http://www.gnu.org/software/cpio/。

```
USER@MACHINE:~/Android$ rm -f ./ramdisk.img.unzip
USER@MACHINE:~/Android$ rm -R ./ramdisk
USER@MACHINE:~/Android$ mv ./ramdisk.img.gz ./out/target/product/generic/ramdisk.img
```

这样，重新打包后得到的ramdisk.img镜像文件中的ueventd.rc文件就修改好了，系统在启动之后就会通过init进程来赋予系统中的所有用户访问设备文件/dev/freg的权限。

2.4 开发Android硬件访问服务

开发好硬件抽象层模块之后，我们通常还需要在应用程序框架层中实现一个硬件访问服务。硬件访问服务通过硬件抽象层模块来为应用程序提供硬件读写操作。由于硬件抽象层模块是使用C++语言开发的，而应用程序框架层中的硬件访问服务是使用Java语言开发的，因此，硬件访问服务必须通过Java本地接口（Java Native Interface，JNI）来调用硬件抽象层模块的接口。

Android系统的硬件访问服务通常运行在系统进程System[1]中，而使用这些硬件访问服务的应用程序运行在另外的进程中，即应用程序需要通过进程间通信机制来访问这些硬件访问服务。Android系统提供了一种高效的进程间通信机制——Binder进程间通信机制[2]，应用程序就是通过它来访问运行在系统进程System中的硬件访问服务的。Binder进程间通信机制要求提供服务的一方必须实现一个具有跨进程访问能力的服务接口，以便使用服务的一方可以通过这个服务接口来访问它。因此，在实现硬件访问服务之前，我们首先要定义它的服务接口。

在本节接下来的内容中，我们将为在2.1小节中介绍的虚拟硬件设备freg开发一个硬件访问服务FregService，它所实现的硬件访问服务接口为IFregService。接下来，我们就首先介绍硬件访问服务接口IFregService的定义，然后介绍硬件访问服务FregService的实现，包括它的内部实现以及调用硬件抽象层模块的JNI实现，最后介绍如何在系统进程System中启动硬件访问服务FregService。

2.4.1 定义硬件访问服务接口

Android系统提供了一种描述语言来定义具有跨进程访问能力的服务接口，这种描述语言称为Android接口描述语言（Android Interface Definition Language，AIDL）。以AIDL定义的服务接口文件是以aidl为后缀名的，在编译时，编译系统会将它们转换成Java文件，然后再对它们进行编译。在本节中，我们将使用AIDL来定义硬件访问服务接口IFregService。

在Android系统中，通常把硬件访问服务接口定义在frameworks/base/core/java/android/os目录中，因此，我们把定义了硬件访问服务接口IFregService的文件IFregService.aidl也保存在这个目录中，它的内容如下所示。

frameworks/base/core/java/android/os/IFregService.aidl

```
1 package android.os;
2
3 interface IFregService {
4         void setVal(int val);
5         int getVal();
6 }
```

IFregService服务接口只定义了两个成员函数，它们分别是setVal和getVal。其中，成员函数setVal用来往虚拟硬件设备freg的寄存器val中写入一个整数，而成员函数getVal用来从虚拟硬件设备freg的寄

1 第11章将详细分析Android系统的系统进程System。

2 第5章将详细分析Android系统的Binder进程间通信机制。

存器val中读出一个整数。

由于服务接口IFregService是使用AIDL语言描述的，因此，我们需要将其添加到编译脚本文件中，这样编译系统才能将其转换为Java文件，然后再对它进行编译。进入到frameworks/base目录中，打开里面的Android.mk文件，修改LOCAL_SRC_FILES变量的值。

```
LOCAL_SRC_FILES += \
    ......
    voip/java/android/net/sip/ISipService.aidl \
    core/java/android/os/IFregService.aidl
```

修改这个编译脚本文件之后，我们就可以使用mmm命令对硬件访问服务接口IFregService进行编译了。

```
USER@MACHINE:~/Android$ mmm ./frameworks/base/
```

编译后得到的framework.jar文件就包含有IFregService接口，它继承了android.os.IInterface接口。在IFregService接口内部，定义了一个Binder本地对象类Stub，它实现了IFregService接口，并且继承了android.os.Binder类。此外，在IFregService.Stub类内部，还定义了一个Binder代理对象类Proxy，它同样也实现了IFregService接口。

前面提到，用AIDL定义的服务接口是用来进行进程间通信的，其中，提供服务的进程称为Server进程，而使用服务的进程称为Client进程。在Server进程中，每一个服务都对应有一个Binder本地对象，它通过一个桩（Stub）来等待Client进程发送进程间通信请求。Client进程在访问运行Server进程中的服务之前，首先要获得它的一个Binder代理对象接口（Proxy），然后通过这个Binder代理对象接口向它发送进程间通信请求。

在接下来的2.4.2小节中，我们就将硬件访问服务FregService从IFregService.Stub类继承下来，并且实现IFregService接口的成员函数setVal和getVal。在2.5小节中，我们再介绍如何在应用程序中获得硬件访问服务FregService的一个Binder代理对象接口，即IFregService.Stub.Proxy接口。

2.4.2　实现硬件访问服务

在Android系统中，通常把硬件访问服务实现在frameworks/base/services/java/com/android/server目录中。因此，我们把实现了硬件访问服务FregService的FregService.java文件也保存在这个目录中，它的内容如下所示。

frameworks/base/services/java/com/android/server/FregService.java

```
01 package com.android.server;
02
03 import android.content.Context;
04 import android.os.IFregService;
05 import android.util.Slog;
06
07 public class FregService extends IFregService.Stub {
08         private static final String TAG = "FregService";
09
10         private int mPtr = 0;
11
12         FregService() {
13                 mPtr = init_native();
14
15                 if(mPtr == 0) {
16                         Slog.e(TAG, "Failed to initialize freg service.");
17                 }
18         }
```

```
19
20          public void setVal(int val) {
21                  if(mPtr == 0) {
22                          Slog.e(TAG, "Freg service is not initialized.");
23                          return;
24                  }
25
26                  setVal_native(mPtr, val);
27          }
28
29          public int getVal() {
30                  if(mPtr == 0) {
31                          Slog.e(TAG, "Freg service is not initialized.");
32                          return 0;
33                  }
34
35                  return getVal_native(mPtr);
36          }
37
38          private static native int init_native();
39          private static native void setVal_native(int ptr, int val);
40          private static native int getVal_native(int ptr);
41  };
```

硬件访问服务FregService继承了IFregService.Stub类，并且实现了IFregService接口的成员函数setVal和getVal。其中，成员函数setVal通过调用JNI方法setVal_native来写虚拟硬件设备freg的寄存器val，而成员函数getVal调用JNI方法getVal_native来读虚拟硬件设备freg的寄存器val。此外，硬件访问服务FregService在启动时，会通过调用JNI方法init_native来打开虚拟硬件设备freg，并且获得它的一个句柄值，保存在成员变量mPtr中。如果硬件访问服务FregService打开虚拟硬件设备freg失败，那么它的成员变量mPtr的值就等于0；否则，就得到一个大于0的句柄值。这个句柄值实际上是指向虚拟硬件设备freg在硬件抽象层中的一个设备对象，硬件访问服务FregService的成员函数 setVal和getVal在访问虚拟硬件设备freg的寄存器val时，必须要指定这个句柄值，以便硬件访问服务FregService的JNI实现可以知道它所要访问的是哪一个硬件设备。

硬件访问服务FregService编写完成之后，就可以执行mmm命令来重新编译Android系统的services模块了。

```
USER@MACHINE:~/Android$ mmm ./frameworks/base/services/java/
```

编译后得到的services.jar文件就包含有FregService类。下面我们继续介绍硬件访问服务FregService的JNI实现。

2.4.3 实现硬件访问服务的JNI方法

在Android系统中，通常把硬件访问服务的JNI方法实现在frameworks/base/services/jni目录中，因此，我们把实现了硬件访问服务FregService的JNI方法的com_android_server_FregService.cpp文件也保存在这个目录中，它的内容如下所示。

frameworks/base/services/jni/com_android_server_FregService.cpp

```
01 #define LOG_TAG "FregServiceJNI"
02
03 #include "jni.h"
04 #include "JNIHelp.h"
05 #include "android_runtime/AndroidRuntime.h"
06
07 #include <utils/misc.h>
```

```
08 #include <utils/Log.h>
09 #include <hardware/hardware.h>
10 #include <hardware/freg.h>
11
12 #include <stdio.h>
13
14 namespace android
15 {
16         /*设置虚拟硬件设备freg的寄存器的值*/
17         static void freg_setVal(JNIEnv* env, jobject clazz, jint ptr, jint value) {
18                 /*将参数ptr转换为freg_device_t结构体变量*/
19                 freg_device_t* device = (freg_device_t*)ptr;
20                 if(!device) {
21                         LOGE("Device freg is not open.");
22                         return;
23                 }
24
25                 int val = value;
26
27                 LOGI("Set value %d to device freg.", val);
28
29                 device->set_val(device, val);
30         }
31
32         /*读取虚拟硬件设备freg的寄存器的值*/
33         static jint freg_getVal(JNIEnv* env, jobject clazz, jint ptr) {
34                 /*将参数ptr转换为freg_device_t结构体变量*/
35                 freg_device_t* device = (freg_device_t*)ptr;
36                 if(!device) {
37                         LOGE("Device freg is not open.");
38                         return 0;
39                 }
40
41                 int val = 0;
42
43                 device->get_val(device, &val);
44
45                 LOGI("Get value %d from device freg.", val);
46
47                 return val;
48         }
49
50         /*打开虚拟硬件设备freg*/
51         static inline int freg_device_open(const hw_module_t* module, struct freg_device_t** device) {
52                 return module->methods->open(module, FREG_HARDWARE_DEVICE_ID, (struct hw_device_t**)device);
53         }
54
55         /*初始化虚拟硬件设备freg*/
56         static jint freg_init(JNIEnv* env, jclass clazz) {
57                 freg_module_t* module;
58                 freg_device_t* device;
59
60                 LOGI("Initializing HAL stub freg......");
61
62                 /*加载硬件抽象层模块freg*/
63                 if(hw_get_module(FREG_HARDWARE_MODULE_ID, (const struct hw_module_t**)&module) == 0) {
64                         LOGI("Device freg found.");
65
66                                 /*打开虚拟硬件设备freg*/
67                                 if(freg_device_open(&(module->common), &device) == 0) {
68                                   LOGI("Device freg is open.");
69
70                                   /*将freg_device_t接口转换为整型值返回*/
```

```
71                                  return (jint)device;
72                          }
73
74                          LOGE("Failed to open device freg.");
75                          return 0;
76                  }
77
78                  LOGE("Failed to get HAL stub freg.");
79
80                  return 0;
81          }
82
83          /*Java本地接口方法表*/
84          static const JNINativeMethod method_table[] = {
85                  {"init_native", "()I", (void*)freg_init},
86                  {"setVal_native", "(II)V", (void*)freg_setVal},
87                  {"getVal_native", "(I)I", (void*)freg_getVal},
88          };
89
90          /*注册Java本地接口方法*/
91          int register_android_server_FregService(JNIEnv *env) {
92                      return jniRegisterNativeMethods(env, "com/android/server/FregService",
method_table, NELEM(method_table));
93          }
94 };
```

在函数freg_init中，首先通过Android硬件抽象层提供的hw_get_module函数来加载模块ID为FREG_HARDWARE_MODULE_ID的硬件抽象层模块。在2.3.3小节中，我们已经介绍过hw_get_module函数是如何根据FREG_HARDWARE_MODULE_ID来加载Android硬件抽象层模块freg的了。函数hw_get_module最终返回一个hw_module_t接口给freg_init函数，这个hw_module_t接口实际指向的是自定义的一个硬件抽象层模块对象，即一个freg_module_t对象。

函数freg_init接着调用函数freg_device_open来打开设备ID为FREG_HARDWARE_DEVICE_ID的硬件设备，而后者又是通过调用前面获得的hw_module_t接口的操作方法列表中的open函数来打开指定的硬件设备的。在2.3.2小节中，我们将硬件抽象层模块freg的操作方法列表中的open函数设置为freg_device_open，这个函数最终返回一个freg_device_t接口给freg_init函数。

函数freg_init最后把获得的freg_device_t接口转换成一个整型句柄值，然后返回给调用者。

函数freg_setVal和freg_getVal都是首先把参数ptr转换为一个freg_device_t接口，然后分别调用它的成员函数set_val和get_val来访问虚拟硬件设备freg的寄存器val的值。

注意

在调用freg_setVal和freg_getVal这两个JNI方法之前，调用者首先要调用JNI方法freg_init打开虚拟硬件设备freg，以便可以获得一个freg_device_t接口。

文件接着定义了一个JNI方法表method_table，分别将函数freg_init、freg_setVal和freg_getVal的JNI方法注册为init_native、setVal_native和getVal_native。文件最后调用了jniRegisterNativeMethods函数把JNI方法表method_table注册到Java虚拟机中，以便提供给硬件访问服务FregService使用。

硬件访问服务FregService的JNI方法编写完成之后，我们还需要修改frameworks/base/services/jni目录下的onload.cpp文件，在里面增加register_android_server_FregService函数的声明和调用。

frameworks/base/services/jni/onload.cpp

```
01 #include "JNIHelp.h"
02 #include "jni.h"
03 #include "utils/Log.h"
04 #include "utils/misc.h"
```

```
05
06 namespace android {
07 ......
08
09 int register_android_server_FregService(JNIEnv* env);
10 };
11
12 using namespace android;
13
14 extern "C" jint JNI_OnLoad(JavaVM* vm, void* reserved)
15 {
16     ......
17
18     register_android_server_FregService(env);
19
20     return JNI_VERSION_1_4;
21 }
```

onload.cpp文件实现在libandroid_servers模块中。当系统加载libandroid_servers模块时，就会调用实现在onload.cpp文件中的JNI_OnLoad函数。这样，就可以将前面定义的三个JNI方法init_native、setVal_native和getVal_native注册到Java虚拟机中。

最后，进入到frameworks/base/services/jni目录中，打开里面的Android.mk文件，修改变量LOCAL_SRC_FILES的值。

```
LOCAL_SRC_FILES += \
    ......
    com_android_server_FregService.cpp \
    onload.cpp
```

修改好这个文件之后，我们就可以执行mmm命令来重新编译libandroid_servers模块了。

```
USER@MACHINE:~/Android$ mmm ./frameworks/base/services/jni/
```

编译后得到的libandroid_servers.so文件就包含有init_native、setVal_native和getVal_native这三个JNI方法了。

至此，硬件访问服务FregService的实现就介绍完了。下面我们继续介绍如何在系统进程System中启动它。

2.4.4　启动硬件访问服务

前面提到，Android系统的硬件访问服务通常是在系统进程System中启动的，而系统进程System是由应用程序孵化器进程Zygote[1]负责启动的。由于应用程序孵化器进程Zygote是在系统启动时启动的，因此，把硬件访问服务运行在系统进程System中，就实现了开机时自动启动。

在本节中，我们把硬件访问服务FregService运行在系统进程System中，因此，进入到frameworks/base/services/java/com/android/server目录中，打开里面的SystemServer.java文件，修改ServerThread类的成员函数run的实现，如下所示。

```
01 class ServerThread extends Thread {
02         ......
03
04         @Override
05           public void run() {
06                 ......
07
```

1　第11章将详细分析应用程序孵化器进程Zygote。

```
08                  if (factoryTest != SystemServer.FACTORY_TEST_LOW_LEVEL) {
09                          ......
10
11                          try {
12                              Slog.i(TAG, "Freg Service");
13                              ServiceManager.addService("freg", new FregService());
14                            } catch (Throwable e) {
15                              Slog.e(TAG, "Failure starting Freg Service", e);
16                            }
17                  }
18
19                  ......
20          }
21
22          ......
23  }
```

系统进程System在启动时，会创建一个ServerThread线程来启动系统中的关键服务，其中就包括一些硬件访问服务。在ServerThread类的成员函数run中，首先创建一个FregService实例，然后把它注册到Service Manager中。Service Manager是Android系统的Binder进程间通信机制的一个重要角色，它负责管理系统中的服务对象。注册到Service Manager中的服务对象都有一个对应的名称，使用这些服务的Client进程就是通过这些名称来向Service Manager请求它们的Binder代理对象接口的，以便可以访问它们所提供的服务。硬件访问服务FregService注册到Service Manager之后，它的启动过程就完成了。

最后，我们需要执行mmm命令来重新编译services模块。

```
USER@MACHINE:~/Android$ mmm ./frameworks/base/services/java/
```

编译后得到的services.jar文件就包含有硬件访问服务FregService，并且在系统启动时，将它运行在系统进程System中。

至此，硬件访问服务FregService就完全实现好了。我们可以执行make snod命令来重新打包Android系统镜像文件system.img。

```
USER@MACHINE:~/Android$ make snod
```

在接下来的2.5小节中，我们将开发一个Android应用程序来访问虚拟硬件设备freg的寄存器val的值，这是通过调用硬件访问服务FregService的成员函数setVal和getVal实现的。

2.5 开发Android应用程序来使用硬件访问服务

在Android应用程序框架层开发硬件访问服务的目的是为了让上层的Android应用程序能够访问对应的硬件设备。在本节中，我们将在Android源代码工程环境中开发一个应用程序Freg，它通过硬件访问服务FregService来访问虚拟硬件设备freg的寄存器val的值。

由于这个应用程序是实验性质的，因此，我们将它放在packages/experimental目录中，对应的工程为Freg。它的目录结构如下：

```
~/Android/packages/experimental/Freg
----AndroidManifest.xml
----Android.mk
----src
    ----shy/luo/freg
        ----Freg.java
----res
    ----layout
        ----main.xml
```

```
----values
    ----strings.xml
----drawable
    ----icon.png
```

它包含一个源代码目录src、一个资源目录res、一个配置文件AndroidManifest.xml和一个编译脚本文件Android.mk。下面我们就分别介绍每一个文件的内容。

Freg.java

```
01 package shy.luo.freg;
02
03 import android.app.Activity;
04 import android.os.ServiceManager;
05 import android.os.Bundle;
06 import android.os.IFregService;
07 import android.os.RemoteException;
08 import android.util.Log;
09 import android.view.View;
10 import android.view.View.OnClickListener;
11 import android.widget.Button;
12 import android.widget.EditText;
13
14 public class Freg extends Activity implements OnClickListener {
15         private final static String LOG_TAG = "shy.luo.freg.FregActivity";
16
17          private IFregService fregService = null;
18
19          private EditText valueText = null;
20          private Button readButton = null;
21          private Button writeButton = null;
22          private Button clearButton = null;
23
24          @Override
25          public void onCreate(Bundle savedInstanceState) {
26                    super.onCreate(savedInstanceState);
27                    setContentView(R.layout.main);
28
29                    fregService = IFregService.Stub.asInterface(
30                              ServiceManager.getService("freg"));
31
32                    valueText = (EditText)findViewById(R.id.edit_value);
33                    readButton = (Button)findViewById(R.id.button_read);
34                    writeButton = (Button)findViewById(R.id.button_write);
35                    clearButton = (Button)findViewById(R.id.button_clear);
36
37                    readButton.setOnClickListener(this);
38                    writeButton.setOnClickListener(this);
39                    clearButton.setOnClickListener(this);
40
41                    Log.i(LOG_TAG, "Freg Activity Created");
42          }
43
44          @Override
45          public void onClick(View v) {
46                    if(v.equals(readButton)) {
47                              try {
48                              int val = fregService.getVal();
49                              String text = String.valueOf(val);
50                              valueText.setText(text);
51                         } catch (RemoteException e) {
52                              Log.e(LOG_TAG, "Remote Exception while reading value from freg service.");
53                         }
```

```
54                    } else if(v.equals(writeButton)) {
55                            try {
56                            String text = valueText.getText().toString();
57                            int val = Integer.parseInt(text);
58                            fregService.setVal(val);
59                        } catch (RemoteException e) {
60                            Log.e(LOG_TAG, "Remote Exception while writing value to freg service.");
61                        }
62                    } else if(v.equals(clearButton)) {
63                        String text = "";
64                        valueText.setText(text);
65                    }
66            }
67 }
```

文件定义了一个Activity组件Freg，它是应用程序Freg的主界面。在Activity组件Freg的界面上，有一个编辑框和三个按钮Read、Write和Clear，其中，编辑框用来显示或者输入虚拟硬件设备freg的寄存器val的值；按钮Read和Write分别用来读写虚拟硬件设备freg的寄存器val，而按钮Clear用来清空编辑框。

在Activity组件Freg的成员函数onCreate中，第30行通过Service Manager获得一个名称为“freg”的服务的Binder代理对象接口。从2.4.4小节的内容可以知道，这个服务就对应于运行在系统进程System中的硬件访问服务FregService。因此，第29行就可以安全地将这个Binder代理对象接口转换为一个FregService代理对象接口，并且保存在Activity组件Freg的成员变量fregService中。有了这个FregService代理对象接口之后，应用程序Freg就可以通过调用它的成员函数setVal和getVal来访问虚拟硬件设备freg的寄存器val的值了，如Activity组件Freg的成员函数onClick所示。

main.xml

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
03     android:orientation="vertical"
04     android:layout_width="fill_parent"
05     android:layout_height="fill_parent"
06     >
07     <LinearLayout
08         android:layout_width="fill_parent"
09         android:layout_height="wrap_content"
10         android:orientation="vertical"
11         android:gravity="center">
12         <TextView
13                 android:layout_width="wrap_content"
14                 android:layout_height="wrap_content"
15                 android:text="@string/value">
16         </TextView>
17         <EditText
18                 android:layout_width="fill_parent"
19                 android:layout_height="wrap_content"
20                 android:id="@+id/edit_value"
21                 android:hint="@string/hint">
22         </EditText>
23     </LinearLayout>
24     <LinearLayout
25         android:layout_width="fill_parent"
26         android:layout_height="wrap_content"
27         android:orientation="horizontal"
28         android:gravity="center">
29         <Button
30                 android:id="@+id/button_read"
31                 android:layout_width="wrap_content"
32                 android:layout_height="wrap_content"
```

```
33                 android:text="@string/read">
34         </Button>
35         <Button
36                 android:id="@+id/button_write"
37                 android:layout_width="wrap_content"
38                 android:layout_height="wrap_content"
39                 android:text="@string/write">
40         </Button>
41         <Button
42                 android:id="@+id/button_clear"
43                 android:layout_width="wrap_content"
44                 android:layout_height="wrap_content"
45                 android:text="@string/clear">
46         </Button>
47     </LinearLayout>
48 </LinearLayout>
```

这是应用程序Freg的主界面配置文件，在屏幕中显示一个TextView控件和三个Button控件。

strings.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3     <string name="app_name">Freg</string>
4     <string name="value">Value</string>
5     <string name="hint">Please input a value...</string>
6     <string name="read">Read</string>
7     <string name="write">Write</string>
8     <string name="clear">Clear</string>
9 </resources>
```

这是应用程序Freg的字符串资源文件，定义了在应用程序中使用到的各个字符串。

icon.png

这是应用程序Freg的图标文件，可以根据需要来放置不同的图片文件。

AndroidManifest.xml

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
03       package="shy.luo.freg"
04       android:versionCode="1"
05       android:versionName="1.0">
06       <application android:icon="@drawable/icon" android:label="@string/app_name">
07            <activity android:name=".Freg"
08                 android:label="@string/app_name">
09                 <intent-filter>
10                      <action android:name="android.intent.action.MAIN" />
11                      <category android:name="android.intent.category.LAUNCHER" />
12                 </intent-filter>
13            </activity>
14       </application>
15 </manifest>
```

这是应用程序Freg的配置文件，由于应用程序Freg定义了一个Activity组件Freg，因此，要在这个配置文件中对它进行配置。

Android.mk

```
1 LOCAL_PATH:= $(call my-dir)
2 include $(CLEAR_VARS)
3 LOCAL_MODULE_TAGS := optional
```

```
4 LOCAL_SRC_FILES := $(call all-subdir-java-files)
5 LOCAL_PACKAGE_NAME := Freg
6 include $(BUILD_PACKAGE)
```

这是应用程序Freg的编译脚本文件，指定程序的名称为“Freg”。

在应用程序Freg的各个文件都准备好以后，就可以对它进行编译和打包了。

```
USER@MACHINE:~/Android$ mmm ./packages/experimental/Freg/
USER@MACHINE:~/Android$ make snod
```

打包后得到的Android系统镜像文件system.img就包含有应用程序Freg了。

最后，我们使用这个新的Android系统镜像文件system.img来启动Android模拟器。

```
USER@MACHINE:~/Android$ emulator -kernel kernel/goldfish/arch/arm/boot/zImage
```

Android模拟器运行起来之后，我们就可以在应用程序启动器中启动应用程序Freg了，它的界面如图2-2所示。

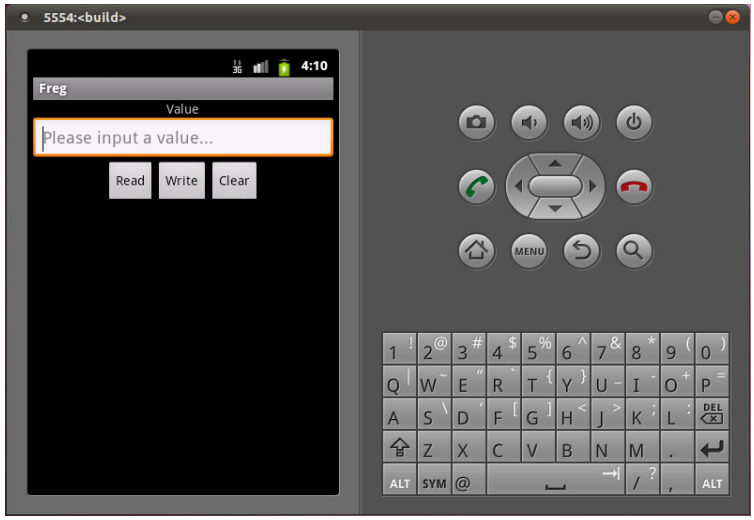


图2-2　应用程序Freg的界面

应用程序Freg启动起来之后，通过点击Read按钮，就可以通过硬件访问服务FregService来读取虚拟硬件设备freg的寄存器val的内容了，而通过在编辑框中输入一个整数值，并且点击Write按钮，就可以通过硬件访问服务FregService将这个整数值写入到虚拟硬件设备freg的寄存器val中了。

第3章 智能指针

在Android系统的应用程序框架层中，有相当一部分代码是使用C++语言开发的。我们知道，用C++来编写代码最容易出错的地方就是指针了，具体表现为忘记释放指针指向的对象所占用的内存，或者使用了无效指针。C++指针使用不当，轻则造成内存泄漏，重则造成莫名其妙的逻辑错误，甚至系统崩溃。既然如此，为什么Android系统的应用程序框架层还要使用C++语言来开发呢？全部都使用Java语言来开发不就可以解决这个问题了吗？出于性能的考虑，有时候还是不得不使用C++代码。既然无法避免使用C++代码，就要想办法避免错误地使用C++指针。因此，Android系统为我们提供了C++智能指针，避免出现指针使用不当的问题。

我们通常通过引用计数技术来维护对象的生命周期。每当有一个新的指针指向了一个对象时，这个对象的引用计数就增加1；相反，每当有一个指针不再指向一个对象时，这个对象的引用计数就减少1；当对象的引用计数为0时，它所占用的内存就可以安全地释放了。这种对象引用计数技术的核心问题是由谁来维护对象的引用计数，如果由开发人员手动地维护对象的引用计数，那么既不可靠，也不方便开发人员编写代码。一旦开发人员一不小心多增加或者多减少了对象的引用计数，那么就会造成灾难性的后果。既然手动维护方式不可取，就必须要找到一种自动的对象引用计数维护技术。

智能指针正是一种能够自动维护对象引用计数的技术。这里需要特别强调的是，智能指针是一个对象，而不是一个指针，但是它引用了一个实际使用的对象。正是因为它是一个对象，因此它能够自动地维护实际对象的引用计数。简单来说，就是在智能指针构造时，增加它所引用的对象的引用计数；而在智能指针析构时，就减少它所引用的对象的引用计数。由于智能指针的构造和析构都是自动的，因此，它就很自然地实现了自动的对象引用计数技术。

这种简单的对象引用计数技术似乎解决了C++指针的问题，然而它是存在缺陷的。考虑这样一个场景：有两个对象A和B，对象A引用了对象B，而对象B也引用了对象A。一方面，当对象A不再使用时，就可以释放它所占用的内存了，但是由于对象B仍然在引用着它，因此，此时对象A就不能被释放；另一方面，当对象B不再使用时，就可以释放它所占用的内存了，但是由于对象A仍然在引用着它，因此，此时对象B也不能被释放。这样问题就出现了，对象A等待对象B被释放，而对象B也在等待对象A被释放。除非能够同时知道对象A和对象B都不再使用了，这个问题才能解决。然而，通常系统是不知道对象之间的依赖关系的，即无法知道两个对象是否同时不再使用。这个问题也是垃圾收集（Garbage Collection）系统所遇到的经典问题之一，因为它一次只能收集一个对象所占用的内存。

这样，我们就需要采取另外一种稍为复杂的引用计数技术来维护对象的生命周期了。这种引用计数技术将对象的引用计数分为强引用计数和弱引用计数两种，其中，对象的生命周期只受强引用计数控制。在使用强引用计数和弱引用计数的解决方案中，一般将有关联的对象划分为“父–子”和“子–父”关系。在“父–子”关系中，“父”对象通过强引用计数来引用“子”对象；而在“子–父”关系中，“子”对象通过弱引用计数来引用“父”对象。这样就可以解决由于相互引用而造成对象

不能释放的问题了。以上面的对象A和B为例，假设对象A和对象B是“父-子”关系，则对象B和对象A就是“子-父”关系。对象A通过强引用计数来引用对象B，而对象B通过弱引用计数来引用对象A。当对象A不再使用时，由于对象B是通过弱引用计数来引用它的，因此，对象A的生命周期不受对象B的影响，此时对象A可以安全地释放。在释放对象A时，同时也会释放它对对象B的强引用计数，因此，当对象B不再使用时，对象B也可以安全地释放了。

由于对象的生命周期不受弱引用计数控制，因此，一方面，当对象B想要使用对象A时，对象A有可能已经被释放了，这时候对象B不能直接使用对象A，而是先要成功地将对象A的弱引用计数升级为强引用计数，然后才能使用它；另一方面，如果对象B不能将对象A的弱引用计数升级为强引用计数，那么就说明对象A已经被释放了，因此，对象B就不能再使用它。

了解了这些背景知识之后，我们就可以开始学习Android系统的智能指针实现原理了。Android系统提供了三种类型的C++智能指针，分别为轻量级指针（Light Pointer）、强指针（Strong Pointer）和弱指针（Weak Pointer），其中，轻量级指针使用了简单的引用计数技术，而强指针和弱指针使用了强引用计数和弱引用计数技术。

无论是轻量级指针，还是强指针或弱指针，它们的实现原理都是类似的，即需要对象提供引用计数器，但是由智能指针来负责维护这个引用计数器。Android系统将引用计数器定义为一个公共类，所有支持使用智能指针的对象类都必须要从这个公共类继承下来。这样，Android系统的智能指针就可以通过这个引用计数器来维护对象的生命周期了。

在本章接下来的内容中，我们首先介绍轻量级指针的实现原理，然后介绍强指针和弱指针的实现原理。

3.1 轻量级指针

前面提到，轻量级指针通过简单的引用计数技术来维护对象的生命周期。如果一个类的对象支持使用轻量级指针，那么它就必须要从LightRefBase类继承下来，因为LightRefBase类提供了一个简单的引用计数器。接下来，我们就先分析轻量级指针的实现原理，然后再通过一个实例介绍它的使用方法。

3.1.1 实现原理分析

我们首先分析LightRefBase类的实现原理，它的定义如下所示。

frameworks/base/include/utils/RefBase.h

```
01 template <class T>
02 class LightRefBase
03 {
04 public:
05     inline LightRefBase() : mCount(0) { }
06     inline void incStrong(const void* id) const {
07         android_atomic_inc(&mCount);
08     }
09     inline void decStrong(const void* id) const {
10         if (android_atomic_dec(&mCount) == 1) {
11             delete static_cast<const T*>(this);
12         }
13     }
14     //! DEBUGGING ONLY: Get current strong ref count.
15     inline int32_t getStrongCount() const {
16         return mCount;
```

```
17      }
18
19  protected:
20      inline ~LightRefBase() { }
21
22  private:
23      mutable volatile int32_t mCount;
24  };
```

LightRefBase类是一个模板类，其中，模板参数T表示对象的实际类型，它必须是继承了LightRefBase类的。LightRefBase类只有一个成员变量mCount，用来描述一个对象的引用计数值。LightRefBase类同时提供了成员函数incStrong和decStrong来增加和减少它所引用的对象的引用计数。

注意

在成员函数decStrong中，如果对象的引用计数值在减少之后变成0，那么就表示需要释放这个对象所占用的内存了。

轻量级指针的实现类为sp，它同时也是强指针的实现类。在本节中，我们只关注它与轻量级指针相关的实现，它的定义如下所示。

frameworks/base/include/utils/RefBase.h

```
01  template <typename T>
02  class sp
03  {
04  public:
05      typedef typename RefBase::weakref_type weakref_type;
06
07      inline sp() : m_ptr(0) { }
08
09      sp(T* other);
10      sp(const sp<T>& other);
11      template<typename U> sp(U* other);
12      template<typename U> sp(const sp<U>& other);
13
14      ~sp();
15
16      // Assignment
17
18      sp& operator = (T* other);
19      sp& operator = (const sp<T>& other);
20
21      template<typename U> sp& operator = (const sp<U>& other);
22      template<typename U> sp& operator = (U* other);
23
24      //! Special optimization for use by ProcessState (and nobody else).
25      void force_set(T* other);
26
27      // Reset
28
29      void clear();
30
31      // Accessors
32
33      inline  T&      operator* () const  { return *m_ptr; }
34      inline  T*      operator-> () const { return m_ptr;  }
35      inline  T*      get() const         { return m_ptr; }
36
37      // Operators
38
39      COMPARE(==)
```

```
40      COMPARE(!=)
41      COMPARE(>)
42      COMPARE(<)
43      COMPARE(<=)
44      COMPARE(>=)
45
46  private:
47      template<typename Y> friend class sp;
48      template<typename Y> friend class wp;
49
50      // Optimization for wp::promote().
51      sp(T* p, weakref_type* refs);
52
53      T*              m_ptr;
54  };
```

sp类也是一个模块类，其中，模板参数T表示对象的实际类型，它也是必须继承了LightRefBase类的。在sp类内部，使用RefBase::weakref_type类来维护它所引用的对象的强引用计数和弱引用计数，在3.2小节中分析强指针的实现原理时，我们再详细分析它。

sp类的实现比较复杂，但是与轻量级指针相关的部分只有成员变量m_ptr以及构造函数和析构函数。不难看出，sp类的成员变量m_ptr是一个指针，它是在构造函数里面初始化的，指向实际引用的对象。接下来我们就分析sp类的构造函数和析构函数的实现。

sp类的构造函数有两个版本，一个是普通的构造函数，一个是拷贝构造函数，如下所示。

frameworks/base/include/utils/RefBase.h

```
01 template<typename T>
02 sp<T>::sp(T* other)
03     : m_ptr(other)
04 {
05     if (other) other->incStrong(this);
06 }
07
08 template<typename T>
09 sp<T>::sp(const sp<T>& other)
10     : m_ptr(other.m_ptr)
11 {
12     if (m_ptr) m_ptr->incStrong(this);
13 }
```

在这两个构造函数里面，首先都是初始化成员变量m_ptr，然后再调用它的成员函数incStrong来增加它的引用计数。由于成员变量m_ptr所指向的对象是从LightRefBase类继承下来的，因此，这两个构造函数实际上是调用了LightRefBase类的成员函数incStrong来增加对象的引用计数。

sp类的析构函数的实现如下所示。

frameworks/base/include/utils/RefBase.h

```
1 template<typename T>
2 sp<T>::~sp()
3 {
4     if (m_ptr) m_ptr->decStrong(this);
5 }
```

sp类的析构函数执行的操作刚好与构造函数相反，即调用成员变量m_ptr所指向的对象的成员函数decStrong来减少它的引用计数，实际上是调用LightRefBase类的成员函数decStrong来减少对象的引用计数。

至此，轻量级指针的实现原理就介绍完了。接下来我们通过一个实例来说明它的使用方法，以

便加深对它的理解。

3.1.2 应用实例分析

我们在external目录中建立一个C++应用程序lightpointer来说明轻量级指针的使用方法，它的目录结构如下：

```
~/Android
----external
     ----lightpointer
          ----lightpointer.cpp
          ----Android.mk
```

应用程序lightpointer的实现很简单，它只有一个源文件 lightpointer.cpp和一个编译脚本文件Android.mk。下面我们就分别介绍它们的内容。

lightpointer.cpp

```
01 #include <stdio.h>
02 #include <utils/RefBase.h>
03
04 using namespace android;
05
06 class LightClass : public LightRefBase<LightClass>
07 {
08 public:
09         LightClass()
10         {
11                 printf("Construct LightClass Object.");
12         }
13
14         virtual ~LightClass()
15         {
16                 printf("Destroy LightClass Object.");
17         }
18 };
19
20 int main(int argc, char** argv)
21 {
22         LightClass* pLightClass = new LightClass();
23         sp<LightClass> lpOut = pLightClass;
24
25         printf("Light Ref Count: %d.\n", pLightClass->getStrongCount());
26
27         {
28                 sp<LightClass> lpInner = lpOut;
29
30                 printf("Light Ref Count: %d.\n", pLightClass->getStrongCount());
31         }
32
33         printf("Light Ref Count: %d.\n", pLightClass->getStrongCount());
34
35         return 0;
36 }
```

这个文件创建了一个继承了LightRefBase类的LightClass类，因此，LightClass类的对象能够结合轻量级指针来使用。

在应用程序lightpointer的入口函数main中，首先创建了一个LightClass对象pLightClass，然后再创建一个轻量级指针lpOut来引用它。在轻量级指针lpOut的创建过程中，会调用sp类的构造函数来增加

LightClass对象pLightClass的引用计数，即此时LightClass对象pLightClass的引用计数值为1，因此，第25行的printf语句打印出来的数字就应该等于1。在第27行到第31行代码中，我们在一个内嵌的作用域中创建了另外一个轻量级指针lpInner来引用LightClass对象pLightClass，此时LightClass对象pLightClass的引用计数值会再增加1，因此，第30行的printf语句打印出来的数字就应该等于2。当应用程序lightpointer跳出了第27行到第31行的作用域之后，轻量级指针lpInner就被析构了，这时候LightClass对象pLightClass的引用计数值就会减少1，因此，第33行的printf语句打印出来的数字就应该等于1。当应用程序lightpointer执行完成之后，轻量级指针lpOut也会被析构，这时候LightClass对象pLightClass的引用计数值就会再次减少1，即等于0，于是LightClass对象pLightClass就会被释放。

Android.mk

```
1 LOCAL_PATH := $(call my-dir)
2 include $(CLEAR_VARS)
3 LOCAL_MODULE_TAGS := optional
4 LOCAL_MODULE := lightpointer
5 LOCAL_SRC_FILES := lightpointer.cpp
6 LOCAL_SHARED_LIBRARIES := \
7             libcutils \
8             libutils
9 include $(BUILD_EXECUTABLE)
```

这是应用程序lightpointer的编译脚本文件，它引用了libcutils和libutils两个库。

上述两个文件准备好了之后，我们就可以对这个C++工程进行编译和打包了。

```
USER@MACHINE:~/Android$ mmm ./external/lightpointer/
USER@MACHINE:~/Android$ make snod
```

编译成功之后，就可以在out/target/product/gerneric/system/bin目录下看到应用程序文件lightpointer了；打包成功之后，该应用程序就包含在out/target/product/gerneric目录下的Android系统镜像文件system.img中了。

最后，我们使用新得到的系统镜像文件system.img来启动Android模拟器。Android模拟器启动起来之后，使用adb工具连接上它，并且进入到/system/bin目录中，运行应用程序lightpointer来查看它的输出，从而验证轻量级指针的实现原理。

```
USER@MACHINE: ~/Android$ emulator &
USER@MACHINE:~/Android$ adb shell
root@android:/ # cd system/bin/
root@android:/system/bin # ./lightpointer
Construct LightClass Object.
Light Ref Count: 1.
Light Ref Count: 2.
Light Ref Count: 1.
Destory LightClass Object.
```

如果能看到上面的输出，就说明我们前面对轻量级指针的实现原理的分析是正确的。

3.2 强指针和弱指针

强指针和弱指针通过强引用计数和弱引用计数来维护对象的生命周期。如果一个类的对象要支持使用强指针和弱指针，那么它就必须从RefBase类继承下来，因为RefBase类提供了强引用计数器和弱引用计数器。

强指针和弱指针的关系比较密切，它们是配合在一起使用的，因此，在本节中，我们把强指针和弱指针放在一起来分析。同样，在本节的最后，我们会通过一个实例来介绍它们的使用方法。

3.2.1 强指针的实现原理分析

我们首先分析RefBase类的实现原理，它的定义如下所示。

frameworks/base/include/utils/RefBase.h

```
01 class RefBase
02 {
03 public:
04     void            incStrong(const void* id) const;
05     void            decStrong(const void* id) const;
06
07     void            forceIncStrong(const void* id) const;
08
09     //! DEBUGGING ONLY: Get current strong ref count.
10     int32_t         getStrongCount() const;
11
12     class weakref_type
13     {
14     public:
15         RefBase*            refBase() const;
16
17         void                incWeak(const void* id);
18         void                decWeak(const void* id);
19
20         bool                attemptIncStrong(const void* id);
21
22         //! This is only safe if you have set OBJECT_LIFETIME_FOREVER.
23         bool                attemptIncWeak(const void* id);
24
25         //! DEBUGGING ONLY: Get current weak ref count.
26         int32_t             getWeakCount() const;
27
28         //! DEBUGGING ONLY: Print references held on object.
29         void                printRefs() const;
30
31         //! DEBUGGING ONLY: Enable tracking for this object.
32         // enable -- enable/disable tracking
33         // retain -- when tracking is enable, if true, then we save a stack trace
34         //           for each reference and dereference; when retain == false, we
35         //           match up references and dereferences and keep only the
36         //           outstanding ones.
37
38         void                trackMe(bool enable, bool retain);
39     };
40
41     weakref_type*   createWeak(const void* id) const;
42
43     weakref_type*   getWeakRefs() const;
44
45     //! DEBUGGING ONLY: Print references held on object.
46     inline  void            printRefs() const { getWeakRefs()->printRefs(); }
47
48     //! DEBUGGING ONLY: Enable tracking of object.
49     inline  void            trackMe(bool enable, bool retain)
50     {
51         getWeakRefs()->trackMe(enable, retain);
52     }
53
54 protected:
55     RefBase();
56     virtual                 ~RefBase();
57
58     //! Flags for extendObjectLifetime()
```

```
59      enum {
60          OBJECT_LIFETIME_WEAK    = 0x0001,
61          OBJECT_LIFETIME_FOREVER = 0x0003
62      };
63
64      void            extendObjectLifetime(int32_t mode);
65
66      //! Flags for onIncStrongAttempted()
67      enum {
68          FIRST_INC_STRONG = 0x0001
69      };
70
71      virtual void            onFirstRef();
72      virtual void            onLastStrongRef(const void* id);
73      virtual bool            onIncStrongAttempted(uint32_t flags, const void* id);
74      virtual void            onLastWeakRef(const void* id);
75
76  private:
77      friend class weakref_type;
78      class weakref_impl;
79
80      RefBase(const RefBase& o);
81      RefBase&        operator=(const RefBase& o);
82
83      weakref_impl* const mRefs;
84  };
```

与LightRefBase类一样，RefBase类也提供了成员函数incStrong和decStrong来维护它所引用的对象的引用计数。不过，RefBase类与LightRefBase类不一样，它不是直接使用一个整数来维护对象的引用计数的，而是使用一个weakref_impl对象，即成员变量mRefs来描述对象的引用计数。

weakref_impl类同时为对象提供了强引用计数和弱引用计数，它的实现如下所示。

frameworks/base/libs/utils/RefBase.cpp

```
01 class RefBase::weakref_impl : public RefBase::weakref_type
02 {
03 public:
04      volatile int32_t    mStrong;
05      volatile int32_t    mWeak;
06      RefBase* const      mBase;
07      volatile int32_t    mFlags;
08
09
10  #if !DEBUG_REFS
11
12      weakref_impl(RefBase* base)
13          : mStrong(INITIAL_STRONG_VALUE)
14          , mWeak(0)
15          , mBase(base)
16          , mFlags(0)
17      {
18      }
19
20      void addStrongRef(const void* /*id*/) { }
21      void removeStrongRef(const void* /*id*/) { }
22      void addWeakRef(const void* /*id*/) { }
23      void removeWeakRef(const void* /*id*/) { }
24      void printRefs() const { }
25      void trackMe(bool, bool) { }
26
27  #else
28      ......
29  #endif
30  };
```

weakref_impl类继承了weakref_type类。weakref_type类定义在RefBase类的内部，它提供了成员函数incWeak、decWeak、attemptIncStrong和attemptIncWeak来维护对象的强引用计数和弱引用计数。weakref_type类只定义了引用计数维护接口，具体的实现是由weakref_impl类提供的。

weakref_impl类有两个成员变量mStrong和mWeak，分别用来描述对象的强引用计数和弱引用计数。同时，weakref_impl类的成员变量mBase指向了它所引用的对象的地址，而成员变量mFlags是一个标志值，用来描述对象的生命周期控制方式。weakref_impl类的成员变量mFlags的取值范围为0、OBJECT_LIFETIME_WEAK或者OBJECT_LIFETIME_FOREVER，其中，0表示对象的生命周期只受强引用计数影响；OBJECT_LIFETIME_WEAK表示对象的生命周期同时受强引用计数和弱引用计数影响；OBJECT_LIFETIME_FOREVER表示对象的生命周期完全不受强引用计数或者弱引用计数影响。

weakref_impl类的实现有调试和非调试两个版本，它们是通过宏DEBUG_REFS来区别的。如果定义了宏DEBUG_REFS，则weakref_impl类被编译成调试版本；否则被编译成非调试版本。weakref_impl类的调试版本只是用于调试它本身的实现，我们不关心，因此省略了这些代码。

下面我们就通过一个类图来总结RefBase、weakref_type和weakref_impl三个类的关系，如图3-1所示。

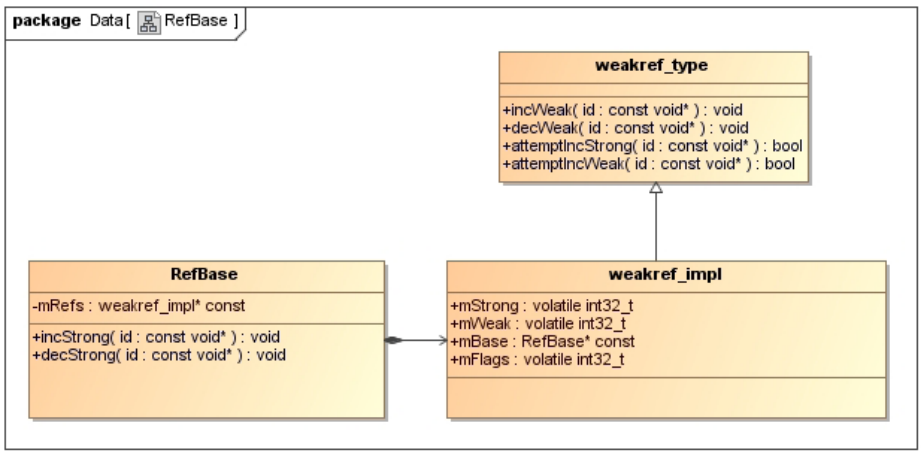


图3-1 RefBase、weakref_type和weakref_impl类的关系

从这个类图就可以看出，每一个RefBase对象都包含了一个weakref_impl对象，而后者继承了weakref_type类。在接下来的内容中，我们会进一步介绍这三个类的作用及其关系。

强指针的实现类为sp，在前面的3.1.1小节中，我们只分析了它的轻量级指针实现。在本节中，我们将分析sp类的强指针实现，主要是分析它的构造函数和析构函数的实现。

sp类的构造函数的实现如下所示。

framew orks/base/include/utils/RefBase.h

```
1 template<typename T>
2 sp<T>::sp(T* other)
3     : m_ptr(other)
4 {
5     if (other) other->incStrong(this);
6 }
```

注意

模块参数T是一个继承了RefBase类的子类，因此，第5行实际上是调用了RefBase类的成员函数incStrong来增加对象的强引用计数，如下所示。

frameworks/base/libs/utils/RefBase.cpp

```
01 void RefBase::incStrong(const void* id) const
02 {
03     weakref_impl* const refs = mRefs;
04     refs->addWeakRef(id);
05     refs->incWeak(id);
06     refs->addStrongRef(id);
07
08     const int32_t c = android_atomic_inc(&refs->mStrong);
09     LOG_ASSERT(c > 0, "incStrong() called on %p after last strong ref", refs);
10
11     #if PRINT_REFS
12     LOGD("incStrong of %p from %p: cnt=%d\n", this, id, c);
13     #endif
14
15     if (c != INITIAL_STRONG_VALUE) {
16         return;
17     }
18
19     android_atomic_add(-INITIAL_STRONG_VALUE, &refs->mStrong);
20     const_cast<RefBase*>(this)->onFirstRef();
21 }
```

RefBase类的成员变量mRefs是在构造函数中初始化的，如下所示。

frameworks/base/libs/utils/RefBase.cpp

```
1 RefBase::RefBase()
2     : mRefs(new weakref_impl(this))
3 {
4 //    LOGV("Creating refs %p with RefBase %p\n", mRefs, this);
5 }
```

回到RefBase类的成员函数incStrong中，它主要做了三件事情：第一件事情是调用第5行代码来增加对象的弱引用计数；第二件事情是调用第8行代码来增加对象的强引用计数；第三件事情是如果发现对象是第一次被强指针引用，则第20行调用对象的成员函数onFirstRef来通知对象，它被强指针引用了，以便它可以执行一些业务相关逻辑。RefBase类的成员函数onFirstRef是一个空实现，如果子类想要处理这个事件，那么就必须要重写成员函数onFirstRef。

增加对象的弱引用计数是通过调用RefBase类的成员变量mRefs的成员函数incWeak来实现的。RefBase类的成员变量mRefs的类型为weakref_impl，它的成员函数incWeak是从父类weakref_type继承下来的，因此，它实际上是通过调用weakref_type类的成员函数incWeak来增加对象的弱引用计数的，如下所示。

frameworks/base/libs/utils/RefBase.cpp

```
1 void RefBase::weakref_type::incWeak(const void* id)
2 {
3     weakref_impl* const impl = static_cast<weakref_impl*>(this);
4     impl->addWeakRef(id);
5     const int32_t c = android_atomic_inc(&impl->mWeak);
6     LOG_ASSERT(c >= 0, "incWeak called on %p after last weak ref", this);
7 }
```

this指针指向的实际上是一个weakref_impl对象，因此，第3行将它转换为一个weakref_impl指针impl。有了这个impl指针之后，我们就可以访问它的成员变量mWeak了。第5行调用函数android_atomic_inc来增加它的成员变量mWeak的值，即增加对象的弱引用计数。第4行的函数调用是与调试相关的，我们忽略它的实现。

回到RefBase类的成员函数incStrong中，增加了对象的弱引用计数之后，接下来就调用函数android_atomic_inc来增加对象的强引用计数了，即增加RefBase类的引用计数对象mRefs的成员变

量mStrong的值。函数android_atomic_inc的返回值是对象原来的强引用计数值，即加1前的值。在weakref_impl类的构造函数中，成员变量mStrong的值被初始化为INITIAL_STRONG_VALUE。INITIAL_STRONG_VALUE是一个宏，它的定义如下所示。

frameworks/base/libs/utils/RefBase.cpp

```
1 #define INITIAL_STRONG_VALUE (1<<28)
```

从理论上说，当对象第一次被强指针引用时，它的强引用计数值应该等于1，但是我们看到，对象的强引用计数的初始值为INITIAL_STRONG_VALUE，它加1之后并不等于1，因此，RefBase类的成员函数incStrong需要将它调整为1，如第19行代码所示。

至此，强指针类sp的构造函数的实现就分析完了，它主要做的事情就是增加对象的强引用计数和弱引用计数。从这里就可以看出，虽然我们的目的是增加对象的强引用计数，但是同时也会增加对象的弱引用计数，即一个对象的弱引用计数一定是大于或者等于它的强引用计数的。

下面我们再来分析sp类的析构函数的实现，如下所示。

frameworks/base/include/utils/RefBase.h

```
1 template<typename T>
2 sp<T>::~sp()
3 {
4     if (m_ptr) m_ptr->decStrong(this);
5 }
```

前面提到，sp类的成员变量m_ptr所指向的对象是继承了RefBase类的，因此，第4行实际上是调用了RefBase类的成员函数decStrong来减少对象的强引用计数，它的实现如下所示。

frameworks/base/libs/utils/RefBase.cpp

```
01 void RefBase::decStrong(const void* id) const
02 {
03     weakref_impl* const refs = mRefs;
04     refs->removeStrongRef(id);
05     const int32_t c = android_atomic_dec(&refs->mStrong);
06 #if PRINT_REFS
07     LOGD("decStrong of %p from %p: cnt=%d\n", this, id, c);
08 #endif
09     LOG_ASSERT(c >= 1, "decStrong() called on %p too many times", refs);
10     if (c == 1) {
11         const_cast<RefBase*>(this)->onLastStrongRef(id);
12         if ((refs->mFlags&OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK) {
13             delete this;
14         }
15     }
16     refs->removeWeakRef(id);
17     refs->decWeak(id);
18 }
```

我们忽略第4行和第16行代码，因为它们在非调试版本中是空函数调用。sp类的析构函数执行的操作刚好与构造函数相反，它主要是减少对象的强引用计数和弱引用计数。

第5行代码通过调用函数android_atomic_dec来减少对象的强引用计数。与函数android_atomic_inc类似，函数android_atomic_dec的返回值是对象原来的强引用计数值，即减1前的值，保存在变量c中。如果变量c的值等于1，就说明此时再也没有强指针引用这个对象了，因此，第11行就调用该对象的成员函数onLastStrongRef来使得它可以执行一些业务相关的逻辑，同时也要考虑是否需要释放该对象。第12行检查对象的生命周期是否受弱引用计数控制，即RefBase类的成员变量mRefs的标志值mFlags的OBJECT_LIFETIME_WEAK位是否等于1。如果不等于1，就说明对象的生命周期不受弱引用计数影响，因此，第13行就会释放对象所占用的内存，这同时会导致RefBase类的析构函数被调用。

frameworks/base/libs/utils/RefBase.cpp

```
1 RefBase::~RefBase()
2 {
3 //    LOGV("Destroying RefBase %p (refs %p)\n", this, mRefs);
4     if (mRefs->mWeak == 0) {
5 //        LOGV("Freeing refs %p of old RefBase %p\n", mRefs, this);
6         delete mRefs;
7     }
8 }
```

在RefBase类的析构函数中，如果发现对象的弱引用计数值为0，那么就会把引用计数对象mRefs也一起释放。RefBase类的成员变量mRefs指向的是一个weakref_impl对象，它是在RefBase类的构造函数中创建的。现在既然它所属的RefBase对象已经不存在了，并且它所引用的对象的弱引用计数值也等于0，它也就不需要存在了。前面提到，一个对象的弱引用计数一定是大于等于它的强引用计数的，因此，当一个对象的强引用计数值为0时，它的弱引用计数值可能大于0。在对象的弱引用计数值大于0的情况下，我们只能将对应的RefBase对象释放掉，而不能将该RefBase对象内部的weakref_impl对象也释放掉，因为还有其他的弱指针通过该weakref_impl对象来引用实际的对象。只有当对象的弱引用计数值也为0 时，才可以将该weakref_impl对象也一起释放掉。

回到RefBase类的成员函数decStrong中，第17行代码执行减少对象的弱引用计数的操作。变量refs指向的是RefBase类内部的weakref_impl对象mRefs。weakref_impl类的成员函数decWeak是从父类weakref_type继承下来的，因此，接下来实际执行的是weakref_type类的成员函数decWeak，它的实现如下所示。

frameworks/base/libs/utils/RefBase.cpp

```
01 void RefBase::weakref_type::decWeak(const void* id)
02 {
03     weakref_impl* const impl = static_cast<weakref_impl*>(this);
04     impl->removeWeakRef(id);
05     const int32_t c = android_atomic_dec(&impl->mWeak);
06     LOG_ASSERT(c >= 1, "decWeak called on %p too many times", this);
07     if (c != 1) return;
08
09     if ((impl->mFlags&OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK) {
10         if (impl->mStrong == INITIAL_STRONG_VALUE)
11             delete impl->mBase;
12         else {
13 //            LOGV("Freeing refs %p of old RefBase %p\n", this, impl->mBase);
14             delete impl;
15         }
16     } else {
17         impl->mBase->onLastWeakRef(id);
18         if ((impl->mFlags&OBJECT_LIFETIME_FOREVER) != OBJECT_LIFETIME_FOREVER) {
19             delete impl->mBase;
20         }
21     }
22 }
```

第4行的函数调用在非调试版本中是空函数调用，我们将它忽略。第5行调用android_atomic_dec函数来减少对象的弱引用计数，并且返回减少之前的值，保存在变量c中。如果c的值不等于1，那么就说明还有其他的弱指针在引用这个对象，因此，就不用进一步处理了；如果c的值等于1，那么就说明再也没有弱指针引用这个对象了，同时也说明没有强指针引用这个对象了，因为当对象的弱引用计数值等于0时，它的强引用计数值也一定等于0。在对象的弱引用计数值等于0时，我们就要考虑是否需要将该对象释放掉。这取决于对象的生命周期控制方式，以及该对象是否被强指针引用过。下面我们分为两种情况来考虑。

第一种情况是对象的生命周期只受强引用计数控制，即第9行的if语句为true。此时如果第10行的if语句也为true，即对象从来没有被强指针引用过，那么在该对象的弱引用计数值等于0时，第11行就需要将该对象释放掉；否则，这个对象以后就会得不到释放。当一个只受强引用计数控制的对象只被弱指针引用时，就会出现这种情况。如果对象的生命周期只受强引用计数控制，并且它也被强指针引用过，那么在该对象的弱引用计数值变为0时，该对象就已经在RefBase类的成员函数decStrong中被释放，因此，第14行只负责释放其内部的引用计数器对象weakref_impl。

第二种情况是对象的生命周期受弱引用计数控制，即第9行的if语句为false。第17行首先调用对象的成员函数onLastWeakRef来使得它可以执行一些业务相关的逻辑，同时也要考虑是否需要将该对象释放掉。第18行检查对象的生命周期是否完全不受强引用计数和弱引用计数控制，即RefBase类的成员变量mRefs的标志值mFlags的OBJECT_LIFETIME_FOREVER位是否等于1。如果不等于1，那么第19行就会释放对象所占用的内存。

从第二种情况就可以看出，当对象的生命周期控制标志值设置为OBJECT_LIFETIME_FOREVER时，即对象的生命周期完全不受强引用计数和弱引用计数控制时，Android系统所提供的智能指针就退化成一个普通的C++指针了，这时候开发人员就需要手动地释放那些不再使用了的对象所占用的内存。

至此，强指针的实现原理就分析完了。为了加深对它的理解，我们对对象的生命周期控制方式作一个小结。

（1）如果一个对象的生命周期控制标志值被设置为0，那么只要它的强引用计数值为0，系统就会自动释放这个对象。

（2）如果一个对象的生命周期控制标志值被设置为OBJECT_LIFETIME_WEAK，那么只有当它的强引用计数值和弱引用计数值都为0时，系统才会自动释放这个对象。

（3）如果一个对象的生命周期控制标志值被设置为OBJECT_LIFETIME_FOREVER，那么系统就永远不会自动释放这个对象，它需要由开发人员来手动地释放。

接下来，我们继续分析弱指针的实现原理。

3.2.2 弱指针的实现原理分析

如果一个类的对象支持使用弱指针，那么这个类就必须要从RefBase类继承下来，因为RefBase类提供了弱引用计数器。在前面的3.2.1小节中，我们已经分析过RefBase类的实现了，因此，在本节中，我们只分析弱指针类wp的实现。

wp类的定义如下所示。

frameworks/base/include/utils/RefBase.h

```
01 template <typename T>
02 class wp
03 {
04 public:
05     typedef typename RefBase::weakref_type weakref_type;
06
07     inline wp() : m_ptr(0) { }
08
09     wp(T* other);
10     wp(const wp<T>& other);
11     wp(const sp<T>& other);
12     template<typename U> wp(U* other);
13     template<typename U> wp(const sp<U>& other);
14     template<typename U> wp(const wp<U>& other);
15
16     ~wp();
```

```
17
18      // Assignment
19
20      wp& operator = (T* other);
21      wp& operator = (const wp<T>& other);
22      wp& operator = (const sp<T>& other);
23
24      template<typename U> wp& operator = (U* other);
25      template<typename U> wp& operator = (const wp<U>& other);
26      template<typename U> wp& operator = (const sp<U>& other);
27
28      void set_object_and_refs(T* other, weakref_type* refs);
29
30      // promotion to sp
31
32      sp<T> promote() const;
33
34      // Reset
35
36      void clear();
37
38      // Accessors
39
40      inline  weakref_type* get_refs() const { return m_refs; }
41
42      inline  T* unsafe_get() const { return m_ptr; }
43
44      // Operators
45
46      COMPARE_WEAK(==)
47          COMPARE_WEAK(!=)
48          COMPARE_WEAK(>)
49          COMPARE_WEAK(<)
50          COMPARE_WEAK(<=)
51          COMPARE_WEAK(>=)
52
53          inline bool operator == (const wp<T>& o) const {
54              return (m_ptr == o.m_ptr) && (m_refs == o.m_refs);
55      }
56      template<typename U>
57      inline bool operator == (const wp<U>& o) const {
58          return m_ptr == o.m_ptr;
59      }
60
61      inline bool operator > (const wp<T>& o) const {
62          return (m_ptr == o.m_ptr) ? (m_refs > o.m_refs) : (m_ptr > o.m_ptr);
63      }
64      template<typename U>
65      inline bool operator > (const wp<U>& o) const {
66          return (m_ptr == o.m_ptr) ? (m_refs > o.m_refs) : (m_ptr > o.m_ptr);
67      }
68
69      inline bool operator < (const wp<T>& o) const {
70          return (m_ptr == o.m_ptr) ? (m_refs < o.m_refs) : (m_ptr < o.m_ptr);
71      }
72      template<typename U>
73      inline bool operator < (const wp<U>& o) const {
74          return (m_ptr == o.m_ptr) ? (m_refs < o.m_refs) : (m_ptr < o.m_ptr);
75      }
76      inline bool operator != (const wp<T>& o) const { return m_refs != o.m_refs; }
77      template<typename U> inline bool operator != (const wp<U>& o) const { return !operator == (o); }
78      inline bool operator <= (const wp<T>& o) const { return !operator > (o); }
79      template<typename U> inline bool operator <= (const wp<U>& o) const { return !operator > (o); }
```

```
80      inline bool operator >= (const wp<T>& o) const { return !operator < (o); }
81      template<typename U> inline bool operator >= (const wp<U>& o) const { return !operator < (o); }
82
83  private:
84      template<typename Y> friend class sp;
85      template<typename Y> friend class wp;
86
87      T*              m_ptr;
88      weakref_type*   m_refs;
89  };
```

wp类是一个模块类，其中，模板参数T表示对象的实际类型，它必须是从RefBase类继承下来的。与强指针类sp类似，弱指针类wp也有一个成员变量m_ptr，用来指向它所引用的对象，但是弱指针类wp还使用另外一个类型为weakref_type*的成员变量m_refs，用来维护对象的弱引用计数。

弱指针与强指针有一个很大的区别，就是弱指针不可以直接操作它所引用的对象，因为它所引用的对象可能是不受弱引用计数控制的，即它所引用的对象可能是一个无效的对象。因此，如果需要操作一个弱指针所引用的对象，那么就需要将这个弱指针升级为强指针，这是通过调用它的成员函数promote来实现的。如果升级成功，就说明该弱指针所引用的对象还没有被销毁，可以正常使用。

wp类的实现比较复杂，但是只要掌握了它的构造函数和析构函数的实现，以及它是如何将一个弱指针升级为强指针的，就可以理解它的实现原理了。因此，在接下来的内容中，我们首先分析wp类的构造函数和析构函数的实现，然后再分析它的成员函数promote的实现。

wp类的构造函数的实现如下所示。

frameworks/base/include/utils/RefBase.h

```
1 template<typename T>
2 wp<T>::wp(T* other)
3     : m_ptr(other)
4 {
5     if (other) m_refs = other->createWeak(this);
6 }
```

注意

模块参数T是一个继承了RefBase类的子类，因此，第5行实际上是调用了RefBase类的成员函数createWeak来增加对象的弱引用计数，如下所示。

frameworks/base/libs/utils/RefBase.cpp

```
1 RefBase::weakref_type* RefBase::createWeak(const void* id) const
2 {
3     mRefs->incWeak(id);
4     return mRefs;
5 }
```

RefBase类的成员变量mRefs指向的是一个weakref_impl对象。在前面的3.2.1小节中，我们已经介绍过它的成员函数incWeak的实现，它主要就是增加实际引用对象的弱引用计数。RefBase类的成员函数createWeak最后将它的成员变量mRefs所指向的一个weakref_impl对象返回给调用者。

wp类的析构函数的实现如下所示。

frameworks/base/include/utils/RefBase.h

```
1 template<typename T>
2 wp<T>::~wp()
3 {
4     if (m_ptr) m_refs->decWeak(this);
5 }
```

弱指针在析构时，会调用它的成员变量m_refs的成员函数decWeak来减少对象的弱引用计数。从

wp类的构造函数的实现可以知道，wp类的成员变量m_refs指向的是一个weakref_impl对象，因此，第4行实际上是调用了weakref_impl类的成员函数decWeak来减少对象的弱引用计数。在前面的3.2.1小节中，我们已经分析过weakref_impl类的成员函数decWeak的实现，它主要就是将对象的弱引用计数值减少1。

介绍完wp类的构造函数和析构函数的实现之后，我们就来重点分析wp类的成员函数promote的实现。因为所有的弱指针都要通过调用这个成员函数来成功升级为强指针之后，才能操作它所引用的对象。wp类是如何使得一个弱指针不能直接操作它所引用的对象的呢？秘密就在于wp类没有重载*和->操作符号，因此，我们就不能直接操作它所引用的对象。回到正题，wp类的成员函数promote的实现如下所示。

frameworks/base/include/utils/RefBase.h

```
1 template<typename T>
2 sp<T> wp<T>::promote() const
3 {
4     return sp<T>(m_ptr, m_refs);
5 }
```

弱指针升级为强指针的方式是通过其内部的成员变量m_ptr和m_refs来创建一个强指针。

frameworks/base/include/utils/RefBase.h

```
1 template<typename T>
2 sp<T>::sp(T* p, weakref_type* refs)
3     : m_ptr((p && refs->attemptIncStrong(this)) ? p : 0)
4 {
5 }
```

参数p指向对象的地址，而参数refs指向该对象内部的一个弱引用计数器对象。只有在对象地址不为NULL的情况下，才会调用它内部的弱引用计数器对象的成员函数attemptIncStrong来试图增加该对象的强引用计数。如果能够成功增加对象的强引用计数，那么就可以成功地把一个弱指针升级为一个强指针了。参数refs是一个类型为weakref_type的指针，因此，接下来就会调用weakref_type类的成员函数attemptIncStrong，它的实现如下所示。

frameworks/base/libs/utils/RefBase.cpp

```
01 bool RefBase::weakref_type::attemptIncStrong(const void* id)
02 {
03     incWeak(id);
04
05     weakref_impl* const impl = static_cast<weakref_impl*>(this);
06
07     int32_t curCount = impl->mStrong;
08     LOG_ASSERT(curCount >= 0, "attemptIncStrong called on %p after underflow",
09         this);
10     while (curCount > 0 && curCount != INITIAL_STRONG_VALUE) {
11         if (android_atomic_cmpxchg(curCount, curCount+1, &impl->mStrong) == 0) {
12             break;
13         }
14         curCount = impl->mStrong;
15     }
16
17     if (curCount <= 0 || curCount == INITIAL_STRONG_VALUE) {
18         bool allow;
19         if (curCount == INITIAL_STRONG_VALUE) {
20             // Attempting to acquire first strong reference...  this is allowed
21             // if the object does NOT have a longer lifetime (meaning the
22             // implementation doesn't need to see this), or if the implementation
23             // allows it to happen.
24             allow = (impl->mFlags&OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK
```

```
25                 || impl->mBase->onIncStrongAttempted(FIRST_INC_STRONG, id);
26         } else {
27             // Attempting to revive the object...  this is allowed
28             // if the object DOES have a longer lifetime (so we can safely
29             // call the object with only a weak ref) and the implementation
30             // allows it to happen.
31             allow = (impl->mFlags&OBJECT_LIFETIME_WEAK) == OBJECT_LIFETIME_WEAK
32                 && impl->mBase->onIncStrongAttempted(FIRST_INC_STRONG, id);
33         }
34         if (!allow) {
35             decWeak(id);
36             return false;
37         }
38         curCount = android_atomic_inc(&impl->mStrong);
39
40         // If the strong reference count has already been incremented by
41         // someone else, the implementor of onIncStrongAttempted() is holding
42         // an unneeded reference.  So call onLastStrongRef() here to remove it.
43         // (No, this is not pretty.)  Note that we MUST NOT do this if we
44         // are in fact acquiring the first reference.
45         if (curCount > 0 && curCount < INITIAL_STRONG_VALUE) {
46             impl->mBase->onLastStrongRef(id);
47         }
48     }
49
50     impl->addWeakRef(id);
51     impl->addStrongRef(id);
52
53 #if PRINT_REFS
54     LOGD("attemptIncStrong of %p from %p: cnt=%d\n", this, id, curCount);
55 #endif
56
57     if (curCount == INITIAL_STRONG_VALUE) {
58         android_atomic_add(-INITIAL_STRONG_VALUE, &impl->mStrong);
59         impl->mBase->onFirstRef();
60     }
61
62     return true;
63 }
```

weakref_type类的成员函数attemptIncStrong试图增加目标对象的强引用计数，但是有可能会增加失败，因为目标对象可能已经被释放了，或者该目标对象不允许使用强指针引用它。

在前面的3.2.1小节中提到，在增加对象的强引用计数时，同时也会增加该对象的弱引用计数，因此，函数第3行首先调用成员函数incWeak来增加对象的弱引用计数。如果后面增加对象的强引用计数失败，则会调用成员函数decWeak来减少对象的弱引用计数。前面提到，wp类的成员变量m_refs实际上指向的是一个weakref_impl对象，因此，第5行可以安全地将this指针转换为一个weakref_impl指针，保存在变量impl中。

一个弱指针所引用的对象可能处于两种状态。第一种状态是该对象同时也正在被其他强指针所引用，即它的强引用计数值大于0，并且不等于INITIAL_STRONG_VALUE；第二种状态是该对象没有被任何强指针引用，即它的强引用计数值小于等于0，或者等于INITIAL_STRONG_VALUE。接下来，我们就根据这两种情况来分析一个弱指针是如何升级为一个强指针的。

第一种情况比较简单，因为这时候对象是一定存在的，因此，我们就可以安全地将这个弱指针升级为强指针。函数第10行到第15行代码处理的正是第一种情况，它将对象的强引用计数值增加1。在增加对象的强引用计数时，要保证原子性，因为其他地方也有可能正在对该对象的强引用计数进行操作。前面我们一般都是直接调用android_atomic_inc函数来保证加1操作的原子性的，但是第11行调用了函数android_atomic_cmpxchg来实现。函数android_atomic_cmpxchg是一个与CPU体系结构相关的函

数，在某些提供了特殊指令的体系结构中，函数android_atomic_cmpxchg执行原子性的加1操作效率会比函数android_atomic_inc高一些。

函数android_atomic_cmpxchg的原型如下所示。

system/core/include/cutils/atomic.h

```
1 int android_atomic_release_cas(int32_t oldvalue, int32_t newvalue,
2         volatile int32_t* addr);
3
4 #define android_atomic_cmpxchg android_atomic_release_cas
```

它只是一个宏定义，实际上指向的是函数android_atomic_release_cas。函数android_atomic_release_cas的工作原理是这样的：如果它发现地址addr的内容等于参数oldvalue的值，那么它就会将地址addr的内容修改为newvalue，然后给调用者返回0，表示修改地址addr的内容成功；否则，就什么也不做，然后给调用者返回1。

回到weakref_type类的成员函数attemptIncStrong中，第11行在调用android_atomic_cmpxchg函数时，将参数oldvalue的值设置为curCount，而将参数newvalue的值设置为curCount + 1。变量curCount保存的是对象的强引用计数，因此，第11行调用android_atomic_cmpxchg函数的目的是增加对象的强引用计数。在什么情况下，函数android_atomic_release_cas会不能成功地增加对象的强引用计数呢？在调用函数android_atomic_cmpxchg之前，对象的强引用计数保存在变量curCount中，在这种情况下，在调用函数android_atomic_cmpxchg时，有可能其他线程已经修改了对象的强引用计数。因此，就会导致函数android_atomic_cmpxchg不能成功地增加对象的强引用计数。如果出现这种情况，第10行到第15行的while循环就必须重新执行增加对象的强引用计数的操作。

在第二种情况下，第17行的if语句会为true，这时候情况就比较复杂了，因为此时对象可能存在，也可能不存在，要进一步进行判断。

如果对象的强引用计数值等于INITIAL_STRONG_VALUE，即第19行的if语句为true，那么就说明这个对象从来没有被强指针引用过。因此，第24行和第25行就根据对象的生命周期控制方式或者对象的实现来判断是否允许将一个引用了它的弱指针升级为强指针。如果对象的生命周期只受强引用计数影响，那么就可以成功地将该弱指针升级为强指针。这一点比较容易理解，因为如果对象的生命周期只受强引用计数影响，而此时该对象又没有被强指针引用过，那么它就必然不会被释放；如果对象的生命周期受弱引用计数影响，即第24行的表达式(impl->mFlags&OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK为false，那么就说明对象现在肯定是存在的，因为现在正有一个弱指针在引用它。但是，这种情况需要进一步调用对象的成员函数onIncStrongAttempted来确认对象是否允许强指针引用它。如果对象的成员函数onIncStrongAttempted的返回值为true，就说明允许使用强指针来引用它，因此，这种情况也可以成功地将该弱指针升级为强指针。RefBase类的成员函数在参数flags为FIRST_INC_STRONG的情况下，是允许将一个指向只受弱引用计数影响生命周期的对象的弱指针升级为一个强指针的，它的实现如下所示。

frameworks/base/libs/utils/RefBase.cpp

```
1 bool RefBase::onIncStrongAttempted(uint32_t flags, const void* id)
2 {
3     return (flags&FIRST_INC_STRONG) ? true : false;
4 }
```

如果对象的强引用计数值小于等于0，那么就会执行第31行和第32行代码。这时候说明对象之前被强指针引用过，因此，就必须要保证对象的生命周期受到弱引用计数的影响；否则，对象就已经被释放了。当对象的生命周期控制标志值的OBJECT_LIFETIME_WEAK位不等于0时，就说明对象的生命周期受到弱引用计数的影响，但是这时候还不能直接将该弱指针升级为强指针，因为对象可能不希

望它被强指针引用。因此，就需要调用对象的成员函数onIncStrongAttempted来进一步确认。如果对象的成员函数onIncStrongAttempted的返回值为true，就说明允许将该弱指针升级为强指针。

在第二种情况下，如果最终得到的变量allow的值为false，那么就说明弱指针升级为强指针失败，于是函数第36行就直接返回false给调用者了。在返回之前，第35行会调用成员函数decWeak来减少对象的弱引用计数，因为函数在开始的地方调用了成员函数incWeak来增加对象的弱引用计数。另外，如果最终得到的变量allow的值为true，那么就说明弱指针可以升级为强指针，因此，第38行就会增加对象的强引用计数。

至此，弱指针升级为强指针的过程就介绍完了，它是弱指针实现原理中最具有挑战性的一个环节。

Android系统提供的强指针和弱指针的实现比较复杂，接下来，我们再通过一个实例来说明它们的使用方法，以便加深对它们的理解。

3.2.3 应用实例分析

我们在external目录下建立一个C++应用程序weightpointer来说明强指针和弱指针的使用方法，它的目录结构如下：

```
~/Android
----external
     ----weightpointer
          ----weightpointer.cpp
          ----Android.mk
```

应用程序weightpointer的实现很简单，它只有一个源文件 weightpointer.cpp和一个编译脚本文件Android.mk。下面我们就分别介绍它们的内容。

weightpointer.cpp

```
001 #include <stdio.h>
002 #include <utils/RefBase.h>
003
004 #define INITIAL_STRONG_VALUE (1<<28)
005
006 using namespace android;
007
008 class WeightClass : public RefBase
009 {
010 public:
011         void printRefCount()
012         {
013                 int32_t strong = getStrongCount();
014                 weakref_type* ref = getWeakRefs();
015
016                 printf("------------------------\n");
017                 printf("Strong Ref Count: %d.\n", (strong  == INITIAL_STRONG_VALUE ? 0 : strong));
018                 printf("Weak Ref Count: %d.\n", ref->getWeakCount());
019                 printf("------------------------\n");
020         }
021 };
022
023 class StrongClass : public WeightClass
024 {
025 public:
026         StrongClass()
027         {
```

```
028                 printf("Construct StrongClass Object.\n");
029         }
030
031         virtual ~StrongClass()
032         {
033                 printf("Destory StrongClass Object.\n");
034         }
035 };
036
037 class WeakClass : public WeightClass
038 {
039 public:
040         WeakClass()
041         {
042                 extendObjectLifetime(OBJECT_LIFETIME_WEAK);
043                 printf("Construct WeakClass Object.\n");
044         }
045
046         virtual ~WeakClass()
047         {
048                 printf("Destory WeakClass Object.\n");
049         }
050 };
051
052 class ForeverClass : public WeightClass
053 {
054 public:
055         ForeverClass()
056         {
057                 extendObjectLifetime(OBJECT_LIFETIME_FOREVER);
058                 printf("Construct ForeverClass Object.\n");
059         }
060
061         virtual ~ForeverClass()
062         {
063                 printf("Destory ForeverClass Object.\n");
064         }
065 };
066
067
068 void TestStrongClass(StrongClass* pStrongClass)
069 {
070         wp<StrongClass> wpOut = pStrongClass;
071         pStrongClass->printRefCount();
072
073         {
074                 sp<StrongClass> spInner = pStrongClass;
075                 pStrongClass->printRefCount();
076         }
077
078         sp<StrongClass> spOut = wpOut.promote();
079         printf("spOut: %p.\n", spOut.get());
080 }
081
082 void TestWeakClass(WeakClass* pWeakClass)
083 {
084         wp<WeakClass> wpOut = pWeakClass;
085         pWeakClass->printRefCount();
086
087         {
088                 sp<WeakClass> spInner = pWeakClass;
089                 pWeakClass->printRefCount();
090         }
```

```
091
092         pWeakClass->printRefCount();
093         sp<WeakClass> spOut = wpOut.promote();
094         printf("spOut: %p.\n", spOut.get());
095 }
096
097 void TestForeverClass(ForeverClass* pForeverClass)
098 {
099         wp<ForeverClass> wpOut = pForeverClass;
100         pForeverClass->printRefCount();
101
102         {
103                 sp<ForeverClass> spInner = pForeverClass;
104                 pForeverClass->printRefCount();
105         }
106 }
107
108 int main(int argc, char** argv)
109 {
110         printf("Test Strong Class: \n");
111         StrongClass* pStrongClass = new StrongClass();
112         TestStrongClass(pStrongClass);
113
114         printf("\nTest Weak Class: \n");
115         WeakClass* pWeakClass = new WeakClass();
116         TestWeakClass(pWeakClass);
117
118         printf("\nTest Forever Class: \n");
119         ForeverClass* pForeverClass = new ForeverClass();
120         TestForeverClass(pForeverClass);
121         pForeverClass->printRefCount();
122         delete pForeverClass;
123
124         return 0;
125 }
```

文件首先定义了一个继承了RefBase类的WeightClass类，它只有一个成员函数printRefCount，用来打印对象的引用计数，包括强引用计数和弱引用计数。

然后又定义了三个类StrongClass、WeakClass和ForeverClass，它们都继承了WeightClass类。其中，StrongClass类对象的生命周期只受强引用计数影响，WeakClass类对象的生命周期同时受到强引用计数和弱引用计数的影响，ForeverClass类对象的生命周期完全不受强引用计数和弱引用计数的影响。

接着又定义了三个测试函数TestStrongClass、TestWeakClass和TestForeverClass，它们分别用来测试强指针和弱指针的三种使用情景，以验证我们对强指针和弱指针的实现原理分析。

在TestStrongClass函数中，第70行将一个StrongClass对象赋值给一个弱指针wpOut，因此，第71行打印出该StrongClass对象的强引用计数值和弱引用计数值应该分别为0和1。接下来第74行再将该StrongClass对象赋值给一个强指针spInner，因此，第75行打印出该StrongClass对象的强引用计数值和弱引用计数值应该分别为1和2。函数执行到第78行时，由于已经超出了强指针spInner的作用域，因此，此时该StrongClass对象的强引用计数值和弱引用计数值应该分别为0和1。由于该StrongClass对象的生命周期只受强引用计数的影响，因此，该StrongClass对象会自动被释放，这一点可以通过观察StrongClass类的析构函数中的日志输出来确认。函数第78行试图将弱指针wpOut升级为强指针，但是由于弱指针wpOut所引用的StrongClass对象已经被释放，因此，弱指针wpOut升级为强指针就会失败，即第79行获得的强指针spOut所引用的对象地址就为0。当TestStrongClass函数返回时，由于超出了弱指针wpOut的作用域，因此，此时该StrongClass对象的弱引用计数值也会减少为0。

在TestWeakClass函数中，第84行将一个WeakClass对象赋值给一个弱指针wpOut，因此，第85

行打印出该WeakClass对象的强引用计数值和弱引用计数值应该分别为0和1。接下来第88行再将该WeakClass对象赋值给一个强指针spInner，因此，第89行打印出该WeakClass对象的强引用计数值和弱引用计数值应该分别为1和2。函数执行到第92行时，由于已经超出了强指针spInner的作用域，因此，此时该WeakClass对象的强引用计数值和弱引用计数值应该为0和1。由于该WeakClass对象的生命周期同时受强引用计数和弱引用计数的影响，因此，此时该WeakClass对象不会被释放。函数第93行试图将弱指针wpOut升级为强指针，由于弱指针wpOut所引用的WeakClass对象还存在，因此，弱指针wpOut就能够成功升级为强指针spOut，即第94行获得的强指针spOut所引用的对象地址就不为0，并且此时该WeakClass对象的强引用计数值和弱引用计数值应该分别为1和2。当TestWeakClass函数返回时，由于超出了弱指针wpOut和强指针spOut的作用域，因此，此时该WeakClass对象的强引用计数值和弱引用计数值都应该减少为0，于是它就会被释放，这一点可以通过观察WeakClass类的析构函数中的日志输出来确认。

在TestForeverClass函数中，第99行将一个ForeverClass对象赋值给一个弱指针wpOut，因此，第100行打印出该ForeverClass对象的强引用计数值和弱引用计数值应该分别为0和1。接下来第103行再将该ForeverClass对象赋值给一个强指针spInner，因此，第104行打印出该ForeverClass对象的强引用计数值和弱引用计数值应该分别为1和2。当TestForeverClass函数返回时，由于超出了弱指针wpOut和强指针spInner的作用域，因此，此时该ForeverClass对象的强引用计数值和弱引用计数值都应该减少为0。但是由于该ForeverClass对象的生命周期不受强引用计数和弱引用计数的影响，因此它不会被自动释放，这一点可以通过观察WeakClass类的析构函数有没有日志输出来确认。

最后，在应用程序weightpointer的入口函数main中，分别调用了上述三个测试函数来验证强指针和弱指针的实现原理。由于第111行和第115行创建的对象都受到引用计数的影响，因此，它们会被自动释放；而第119行创建的对象不受引用计数的影响，因此，我们需要在第122行中手动地释放该对象。

Android.mk

```
1 LOCAL_PATH := $(call my-dir)
2 include $(CLEAR_VARS)
3 LOCAL_MODULE_TAGS := optional
4 LOCAL_MODULE := weightpointer
5 LOCAL_SRC_FILES := weightpointer.cpp
6 LOCAL_SHARED_LIBRARIES :=  \
7         libcutils \
8         libutils
9 include $(BUILD_EXECUTABLE)
```

这是应用程序weightpointer的编译脚本文件，它引用了libcutils和libutils两个库。

上述两个文件准备好之后，我们就可以对这个C++工程进行编译和打包了。

```
USER@MACHINE:~/Android$ mmm ./external/weightpointer/
USER@MACHINE:~/Android$ make snod
```

编译成功之后，就可以在out/target/product/gerneric/system/bin目录下看到应用程序文件weightpointer了；打包成功之后，该应用程序就包含在out/target/product/gerneric目录下的Android系统镜像文件system.img中了。

最后，我们使用新得到的系统镜像文件system.img来启动Android模拟器。Android模拟器启动起来之后，使用adb工具连接上它，并进入到/system/bin目录中，运行应用程序weightpointer来查看它的输出，从而验证强指针和弱指针的实现原理。

```
USER@MACHINE: ~/Android$ emulator &
USER@MACHINE:~/Android$ adb shell
root@android:/ # cd system/bin/
root@android:/system/bin # ./weightpointer
Test Strong Class:
Construct StrongClass Object.
-----------------------
Strong Ref Count: 0.
Weak Ref Count: 1.
-----------------------
-----------------------
Strong Ref Count: 1.
Weak Ref Count: 2.
-----------------------
Destory StrongClass Object.
spOut: 0x0.

Test Weak Class:
Construct WeakClass Object.
-----------------------
Strong Ref Count: 0.
Weak Ref Count: 1.
-----------------------
-----------------------
Strong Ref Count: 1.
Weak Ref Count: 2.
-----------------------
-----------------------
Strong Ref Count: 0.
Weak Ref Count: 1.
-----------------------
spOut: 0xa528.
Destory WeakClass Object.

Test Forever Class:
Construct ForeverClass Object.
-----------------------
Strong Ref Count: 0.
Weak Ref Count: 1.
-----------------------
-----------------------
Strong Ref Count: 1.
Weak Ref Count: 2.
-----------------------
-----------------------
Strong Ref Count: 0.
Weak Ref Count: 0.
-----------------------
Destory ForeverClass Object.
```

如果能看到上面的输出，就说明我们前面对强指针和弱指针的实现原理的分析是正确的。

第2篇　Android专用驱动系统

Android系统是基于Linux内核开发的，但是为了更好地支持移动设备，它在Linux内核中增加了一些专用的驱动程序。这些专用的驱动程序形成了整个Android系统的坚实基础，它们被广泛地应用在Android系统的应用程序框架层中。正是因为这些专用的驱动程序有着如此重要的作用，我们就有必要重点掌握它们的实现原理。

Android系统在Linux内核中增加的驱动程序比较多，但是我们在阅读Android系统的应用程序框架层源代码时，经常碰到的是其中的三个驱动程序，分别为Logger日志驱动程序、Binder进程间通信驱动程序和Ashmem匿名共享内存驱动程序。在本篇中，我们将使用三个章节的篇幅来分析它们的实现原理。

第4章分析Logger日志系统的实现原理。在应用程序的开发过程中，我们经常使用日志来调试应用程序的功能；而在应用程序的运营过程中，我们同样需要使用日志来记录运营事件，以便可以对运营问题进行跟踪处理或者对运营数据进行统计等。正是因为日志有着如此重要的作用，开发人员应该习惯性地在系统的关键路径中添加日志记录。Android系统提供了一个日志系统，它是基于内核中的Logger日志驱动程序来实现的，并且在应用程序框架层中提供了写日志的Java和C++接口。此外，Android系统还提供了一个用来读取日志记录的实用工具Logcat，开发人员通过Logcat可以方便地获取系统运行过程中的日志输出。在第4章中，我们将分别介绍Logger日志驱动程序、应用程序框架层中的日志写入接口，以及日志读取工具Logcat的实现原理。

第5章分析Binder进程间通信机制的实现原理。Android应用程序是由Activity、Service、Content Provider和Broadcast Receiver四大组件组成的。为了降低应用程序各个模块之间的耦合性，我们可以把这些应用程序组件运行在独立的进程中，这样就需要有一种高效的进程间通信机制。Android系统提供的Binder进程间通信机制正是一种高效的进程间通信机制。Binder进程间通信机制由Binder进程间通信驱动程序、Service Manager、Client和Service四个角色组成。其中，Service负责提供服务，Client向Service请求服务，Service Manager负责管理系统中的Service，同时负责向Client提供Service查询服务，而Binder进程间通信驱动程序负责在Client和Service之间建立进程间通信通道。在第5章中，我们将分别介绍Binder进程间通信驱动程序、Service Manager、Client和Service的实现原理。

第6章分析Ashmem匿名共享内存机制的实现原理。当两个进程需要共享数据时，我们通常可以通过两种方式来实现。第一种方式是在进程间复制共享数据，即将共享数据在进程间传输；第二种方式是将共享数据放在一个公共的地方，然后每一个进程都到这个公共的地方去访问共享数据。如果进程间需要共享的数据很大，那么显然第二种共享方式的效率更高。因此，Android系统提供了一种Ashmem匿名共享内存机制来在进程间进行数据共享，它通常结合Binder进程间通信机制来使用。Ashmem匿名共享内存机制是基于Ashmem匿名共享内存驱动程序实现的，它可以协助内存管理系统来高效地管理系统内存。在第6章中，我们将分别介绍Ashmem匿名共享内存机制的驱动程序实现、进程间数据共享原理，以及Java、C++访问接口。

第4章 Logger日志系统

Android提供的Logger日志系统是基于内核中的Logger日志驱动程序实现的，它将日志记录保存在内核空间中。为了有效地利用内存空间，Logger日志驱动程序在内部使用一个环形缓冲区来保存日志。因此，当Logger日志驱动程序中的环形缓冲区满了之后，新的日志就会覆盖旧的日志。

由于新的日志会覆盖旧的日志，因此，Logger日志驱动程序根据日志的类型以及日志的输出量来对日志记录进行分类，避免重要的日志被不重要的日志覆盖，或者数据量大的日志覆盖数据量小的日志。日志的类型一共有四种，它们分别是main、system、radio和events。在Logger日志驱动程序中，这四种类型的日志分别通过/dev/log/main、/dev/log/system、/dev/log/radio和/dev/log/events四个设备文件来访问。

类型为main的日志是应用程序级别的，而类型为system的日志是系统级别的。由于系统级日志要比应用程序级日志重要，因此，把它们分开来记录，可以避免系统级日志被应用程序级日志覆盖。类型为radio的日志是与无线设备相关的，它们的量很大，因此，把它们单独记录在一起，可以避免它们覆盖其他类型的日志。类型为events的日志是专门用来诊断系统问题的，应用程序开发者不应该使用此种类型的日志。

Android系统在应用程序框架层中提供了android.util.Log、android.util.Slog和android.util.EventLog三个Java接口来往Logger日志驱动程序中写入日志，它们写入的日志类型分别为main、system和events。特别地，如果使用android.util.Log和android.util.Slog接口写入的日志的标签值是以“RIL”开头或者等于“HTC_RIL”、“AT”、“GSM”、“STK”、“CDMA”、“PHONE”和“SMS”时，它们就会被转换为radio类型的日志写入到Logger日志驱动程序中。相应地，Android系统在运行时库层也提供了三组C/C++宏来往Logger日志驱动程序中写入日志。其中，宏LOGV、LOGD、LOGI、LOGW和LOGE用来写入main类型的日志，宏SLOGV、SLOGD、SLOGI、SLOGW和SLOGE用来写入system类型的日志，宏LOG_EVENT_INT、LOG_EVENT_LONG和LOG_EVENT_STRING用来写入events类型的日志。无论是Java日志写入接口还是C/C++日志写入接口，它们最终都是通过运行时库层的日志库liblog来往Logger日志驱动程序中写入日志的。此外，系统还提供了一个Logcat工具来读取和显示Logger日志驱动程序中的日志。

以上介绍的Logger日志系统框架可以通过图4-1来形象地描述。

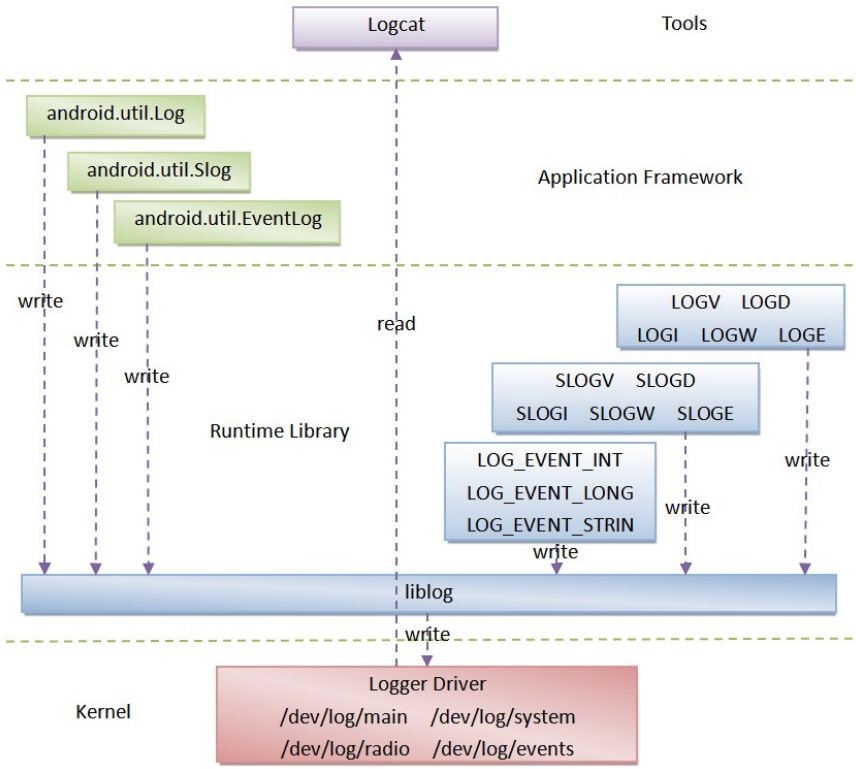


图4–1 Logger日志系统框架

在本章内容中，我们首先介绍Logger日志系统的日志格式，然后分析Logger日志驱动程序、运行时库层的日志库liblog、C/C++和Java日志写入接口的实现，最后分析日志读取工具Logcat的实现。

4.1 Logger日志格式

前面提到，Logger日志一共划分为main、system、radio和events四种类型，其中，前面三种类型的日志格式是相同的，而第四种类型的日志格式稍有区别。下面我们就分别介绍它们的格式。

类型为main、system和radio的日志格式如图4–2所示。

priority	tag	msg

图4–2 类型为main、system和radio的日志格式

其中，priority表示日志优先级，它是一个整数；tag表示日志标签，它是一个字符串；msg表示日志内容，它也是一个字符串。日志优先级和日志标签可以在显示日志时作过滤字段使用。日志优先级按照重要程度一般划分为VERBOSE、DEBUG、INFO、WARN、ERROR和FATAL六种。

类型为events的日志格式如图4–3所示。

tag	msg

图4–3 类型为events的日志格式

其中，tag表示日志标签，它是一个整数；msg表示日志内容，它是一块二进制数据，它的内容格式是由日志写入者来决定的。一般来说，这些日志内容是由一个或者多个值组成的，每个值的前面都有一个字段来描述它的类型，如图4–4所示。

type of value 1	value 1	type of value 2	value 2	

图4-4　类型为events的日志内容的格式

其中，值的类型为整数（int）、长整数（long）、字符串（string）或者列表（list），它们分别使用数字1、2、3和4来描述。

由于events类型的日志标签是一个整数值，在显示时不具有可读性，因此，Android系统使用设备上的日志标签文件/system/etc/event-log-tags来描述这些标签值的含义。这样，Logcat工具在显示events类型的日志时，就可以把日志中的标签值转换为字符串。此外，该日志标签文件还用来描述events类型的日志内容的格式，它的格式如图4-5所示。

tag number	tag name	format for tag value

图4-5　日志标签文件格式

其中，第一个字段tag number表示日志标签值，它的取值范围为0～2147483648；第二个字段tag name是日志标签值对应的字符串描述，它是由字母[A-Z][a-z]、数字[0-9]或者下画线“_”组成的；第三个字段用来描述组成日志内容的值格式，如图4-6所示。

(name\|data type[\|data unit])[,(name\|data type[\|data unit]),.....]

图4-6　类型为events的日志内容的值格式

其中，name表示日志内容值的名称；data type表示日志内容值的数据类型，它的取值范围为1～4，分别表示整数（int）、长整数（long）、字符串（string）和列表（list）；data unit表示日志内容值的数据单位，它的取值范围是1～6，分别表示对象数量（number of objects）、字节数（Number of bytes）、毫秒数（Number of milliseconds）、分配额（Number of allocations）、标志（ID）和百分比（Percent）。

最后，我们从Android模拟器上的日志标签文件/system/etc/event-log-tags中取出一行内容来说明类型为events的日志格式，如下所示。

```
2722 battery_level (level|1|6),(voltage|1|1),(temperature|1|1)
```

其中，2722表示日志标签值，而battery_level用来描述日志标签值2722的含义。此外，从battery_level后面的内容可以看出，日志标签值等于2722的日志内容由三个值组成，它们分别是level、voltage和temperature，对应的数据类型都是整数（int），但是level的单位是百分比（6），voltage和temperature的单位均为对象数量（1）。

4.2　Logger日志驱动程序

Logger日志驱动程序实现在系统的内核空间中，它的目录结构如下：

```
~/Android/kernel/goldfish
----drivers
     ----staging
          ----android
               ----logger.h
               ----logger.c
```

它主要由logger.h和logger.c两个源文件组成。在本节接下来的内容中，我们首先介绍Logger日志驱动程序的基础数据结构，然后分别介绍日志设备的初始化、打开、读取和写入过程。

4.2.1 基础数据结构

Logger日志驱动程序主要使用到了logger_entry、logger_log和logger_reader三个结构体。下面我们就详细描述它们的定义。

struct logger_entry

kernel/goldfish/drivers/staging/android/logger.h

```
01 struct logger_entry {
02         __u16           len;    /* length of the payload */
03         __u16           __pad;  /* no matter what, we get 2 bytes of padding */
04         __s32           pid;    /* generating process's pid */
05         __s32           tid;    /* generating process's tid */
06         __s32           sec;    /* seconds since Epoch */
07         __s32           nsec;   /* nanoseconds */
08         char            msg[0]; /* the entry's payload */
09 };
10
11 #define LOGGER_ENTRY_MAX_LEN            (4*1024)
12 #define LOGGER_ENTRY_MAX_PAYLOAD        \
13         (LOGGER_ENTRY_MAX_LEN - sizeof(struct logger_entry))
```

结构体logger_entry用来描述一个日志记录。每一个日志记录的最大长度为4K，其中，日志记录的有效负载长度最大等于4K减去结构体logger_entry的大小。成员变量len表示实际的日志记录的有效负载长度；成员变量__pad没有实际意义，是用来将成员变量pid的地址值对齐到4字节边界的；成员变量pid和tid记录写入该日志记录的进程的TGID和PID，即线程组ID和进程ID；成员变量sec和nsec记录写入该日志记录的时间，它们是用标准基准时间（Epoch）[1]来描述的；成员变量msg记录的是实际写入的日志记录内容，它的长度是可变的，由成员变量len决定。

struct logger_log

kernel/goldfish/drivers/staging/android/logger.c

```
01 /*
02  * struct logger_log - represents a specific log, such as 'main' or 'radio'
03  *
04  * This structure lives from module insertion until module removal, so it does
05  * not need additional reference counting. The structure is protected by the
06  * mutex 'mutex'.
07  */
08 struct logger_log {
09         unsigned char *         buffer; /* the ring buffer itself */
10         struct miscdevice       misc;   /* misc device representing the log */
11         wait_queue_head_t       wq;     /* wait queue for readers */
12         struct list_head        readers; /* this log's readers */
13         struct mutex            mutex;  /* mutex protecting buffer */
14         size_t                  w_off;  /* current write head offset */
15         size_t                  head;   /* new readers start here */
16         size_t                  size;   /* size of the log */
17 };
```

结构体logger_log用来描述一个日志缓冲区。在Logger日志驱动程序中，每一种类型的日志都对应有一个单独的日志缓冲区。成员变量buffer指向一个内核缓冲区，用来保存日志记录，它是循环使用的，大小由成员变量size决定；成员变量misc的类型为miscdevice，用来描述一个日志设备[2]；成员变量

1 也称为格林威治时间或者新纪元时间，即1970.1.1 00:00:00 GMT。

2 Linux将硬件设备划分为字符（char）设备、块（block）设备和misc设备三种类型。每一个设备都有自己的主设备号和从设备号，其中，相同类型的设备具有相同的主设备号，但是具有不同的从设备号。有关Linux设备的知识，可以参考*Linux Device Drivers*一书。

wq是一个等待队列，用来记录那些正在等待读取新的日志记录的进程；成员变量readers是一个列表，用来记录那些正在读取日志记录的进程，每一个进程都使用一个结构体logger_reader来描述；成员变量mutex是一个互斥量，用来保护日志缓冲区的并发访问[1]；成员变量w_off表示下一条要写入的日志记录在日志缓冲区中的位置；成员变量head表示当一个新的进程读取日志时，它应该从日志缓冲区中的什么位置开始读取。

struct logger_reader

kernel/goldfish/drivers/staging/android/logger.c

```
01 /*
02  * struct logger_reader - a logging device open for reading
03  *
04  * This object lives from open to release, so we don't need additional
05  * reference counting. The structure is protected by log->mutex.
06  */
07 struct logger_reader {
08         struct logger_log *      log;    /* associated log */
09         struct list_head         list;   /* entry in logger_log's list */
10         size_t                   r_off;  /* current read head offset */
11 };
```

结构体logger_reader用来描述一个正在读取某一个日志缓冲区的日志记录的进程。成员变量log指向要读取的日志缓冲区结构体；成员变量list是一个列表项，用来连接所有读取同一种类型日志记录的进程；成员变量r_off表示当前进程要读取的下一条日志记录在日志缓冲区中的位置。

4.2.2 日志设备的初始化过程

日志设备的初始化过程是在Logger日志驱动程序的入口函数logger_init中进行的。在分析这个函数之前，我们首先介绍三个结构体变量log_main、log_events和log_radio，它们的类型均为struct logger_log，分别用来保存main、events和radio三种类型的日志记录，如下所示。

kernel/goldfish/drivers/staging/android/logger.c

```
01 /*
02  * Defines a log structure with name 'NAME' and a size of 'SIZE' bytes, which
03  * must be a power of two, greater than LOGGER_ENTRY_MAX_LEN, and less than
04  * LONG_MAX minus LOGGER_ENTRY_MAX_LEN.
05  */
06 #define DEFINE_LOGGER_DEVICE(VAR, NAME, SIZE) \
07 static unsigned char _buf_ ## VAR[SIZE]; \
08 static struct logger_log VAR = { \
09         .buffer = _buf_ ## VAR, \
10         .misc = { \
11                 .minor = MISC_DYNAMIC_MINOR, \
12                 .name = NAME, \
13                 .fops = &logger_fops, \
14                 .parent = NULL, \
15         }, \
16         .wq = __WAIT_QUEUE_HEAD_INITIALIZER(VAR .wq), \
17         .readers = LIST_HEAD_INIT(VAR .readers), \
18         .mutex = __MUTEX_INITIALIZER(VAR .mutex), \
19         .w_off = 0, \
20         .head = 0, \
21         .size = SIZE, \
22 };
23
```

1 同一种类型的日志可能会被多个进程同时读写，因此，需要对它的日志缓冲区进行并发访问保护。

```
24 DEFINE_LOGGER_DEVICE(log_main, LOGGER_LOG_MAIN, 64*1024)
25 DEFINE_LOGGER_DEVICE(log_events, LOGGER_LOG_EVENTS, 256*1024)
26 DEFINE_LOGGER_DEVICE(log_radio, LOGGER_LOG_RADIO, 64*1024)
```

结构体变量log_main、log_events和log_radio都是通过宏DEFINE_LOGGER_DEVICE来定义的，需要传入的三个参数分别为变量名、设备文件路径，以及要分配的日志缓冲区的大小。前面提到，Logger日志系统将日志记录划分为四种类型。从理论上说，每一种类型的日志记录都应该对应有一个日志缓冲区，即在Logger日志驱动程序中对应有一个logger_log结构体，但是在目前的Logger日志驱动程序实现中，只定义了三个日志缓冲区，其中，类型为system和main的日志记录保存在同一个日志缓冲区log_main中，而类型为events和radio的日志记录分别保存在日志缓冲区log_events和log_radio中。

从宏DEFINE_LOGGER_DEVICE的定义可以看出，每一种类型的日志缓冲区都是将各自的日志记录保存在一个静态分配的unsigned char数组中，数组的大小由参数SIZE决定，其中，类型为main和radio的日志缓冲区的大小为64K，而类型为events的日志缓冲区的大小为256K。

每一个日志缓冲区都对应有一个misc类型的日志设备，以便运行在用户空间的程序可以访问它们。每一个日志设备都对应有一个文件名，它是由参数NAME指定的。从结构体变量log_main、log_events和log_radio的定义可以看出，类型为main、radio和events的日志缓冲区对应的日志设备文件分别为LOGGER_LOG_MAIN、LOGGER_LOG_RADIO和LOGGER_LOG_EVENTS。

kernel/goldfish/drivers/staging/android/logger.h

```
1 #define LOGGER_LOG_RADIO        "log_radio"     /* radio-related messages */
2 #define LOGGER_LOG_EVENTS       "log_events"    /* system/hardware events */
3 #define LOGGER_LOG_MAIN         "log_main"      /* everything else */
```

Logger日志驱动程序初始化完成之后，我们就可以在/dev/log目录下看到三个日志设备文件main、events和radio。这三个日志设备对应的文件操作函数表均为logger_fops，它的定义如下所示。

kernel/goldfish/drivers/staging/android/logger.c

```
01 static struct file_operations logger_fops = {
02         .owner = THIS_MODULE,
03         .read = logger_read,
04         .aio_write = logger_aio_write,
05         .poll = logger_poll,
06         .unlocked_ioctl = logger_ioctl,
07         .compat_ioctl = logger_ioctl,
08         .open = logger_open,
09         .release = logger_release,
10 };
```

文件操作函数表logger_fops指定的日志设备文件操作函数比较多，但是我们只关注日志设备打开、读取和写入函数open、read和aio_write的定义，它们分别被设置为logger_open、logger_read和logger_aio_write函数。

每一个硬件设备都有自己的主设备号和从设备号。由于日志设备的类型为misc，并且同一类型的设备都具有相同的主设备号，因此，在定义这些日志设备时，就不需要指定它们的主设备号了，等到注册这些日志设备时，再统一指定。从宏DEFINE_LOGGER_DEVICE的定义可以看出，这些日志设备的从设备号被设置为MISC_DYNAMIC_MINOR，表示由系统动态分配，它的定义如下所示。

kernel/goldfish/include/linux/miscdevice.h

```
1 #define MISC_DYNAMIC_MINOR      255
```

一般来说，我们在开发驱动程序时，都不应该去静态指定设备的从设备号，因为静态指定的从设备号会比较容易发生冲突，而动态分配的从设备号能够保证唯一性。

结构体变量log_main、log_events和log_radio的其余成员变量的初始化过程都比较简单，这里就不详细介绍了。例如，它们的成员变量wq、reader和mutex分别使用宏__WAIT_QUEUE_HEAD_

INITIALIZER、LIST_HEAD_INIT和__MUTEX_INITIALIZER来初始化，因为它们的类型分别为等待队列、队列和互斥量。

我们就从函数logger_init开始，分析日志设备的初始化过程。

kernel/goldfish/drivers/staging/android/logger.c

```
01 static int __init logger_init(void)
02 {
03         int ret;
04
05         ret = init_log(&log_main);
06         if (unlikely(ret))
07                 goto out;
08
09         ret = init_log(&log_events);
10         if (unlikely(ret))
11                 goto out;
12
13         ret = init_log(&log_radio);
14         if (unlikely(ret))
15                 goto out;
16
17 out:
18         return ret;
19 }
```

日志设备的初始化过程实际上就是将三个日志设备注册到系统中，这是分别通过调用函数init_log来实现的。

kernel/goldfish/drivers/staging/android/logger.c

```
01 static int __init init_log(struct logger_log *log)
02 {
03         int ret;
04
05         ret = misc_register(&log->misc);
06         if (unlikely(ret)) {
07                 printk(KERN_ERR "logger: failed to register misc "
08                        "device for log '%s'!\n", log->misc.name);
09                 return ret;
10         }
11
12         printk(KERN_INFO "logger: created %luK log '%s'\n",
13                (unsigned long) log->size >> 10, log->misc.name);
14
15         return 0;
16 }
```

在init_log函数中，主要是通过调用misc_register函数将相应的日志设备注册到系统中。

kernel/goldfish/drivers/char/misc.c

```
01 int misc_register(struct miscdevice * misc)
02 {
03         struct miscdevice *c;
04         dev_t dev;
05         int err = 0;
06
07         INIT_LIST_HEAD(&misc->list);
08
09         mutex_lock(&misc_mtx);
10         list_for_each_entry(c, &misc_list, list) {
11                 if (c->minor == misc->minor) {
12                         mutex_unlock(&misc_mtx);
13                         return -EBUSY;
```

```
14                 }
15         }
16
17         if (misc->minor == MISC_DYNAMIC_MINOR) {
18                 int i = DYNAMIC_MINORS;
19                 while (--i >= 0)
20                         if ( (misc_minors[i>>3] & (1 << (i&7))) == 0)
21                                 break;
22                 if (i<0) {
23                         mutex_unlock(&misc_mtx);
24                         return -EBUSY;
25                 }
26                 misc->minor = i;
27         }
28
29         if (misc->minor < DYNAMIC_MINORS)
30                 misc_minors[misc->minor >> 3] |= 1 << (misc->minor & 7);
31         dev = MKDEV(MISC_MAJOR, misc->minor);
32
33         misc->this_device = device_create(misc_class, misc->parent, dev, NULL,
34                                           "%s", misc->name);
35         if (IS_ERR(misc->this_device)) {
36                 err = PTR_ERR(misc->this_device);
37                 goto out;
38         }
39
40         /*
41          * Add it to the front, so that later devices can "override"
42          * earlier defaults
43          */
44         list_add(&misc->list, &misc_list);
45  out:
46         mutex_unlock(&misc_mtx);
47         return err;
48 }
```

系统中所有的misc设备都保存在一个misc_list列表中。函数第10行到第15行检查所要注册的misc设备的从设备号是否已经被注册。如果是的话，注册就失败了，因为同一类型设备的从设备号是不能相同的。由于Logger日志驱动程序已经指定它所要注册的日志设备的从设备号是动态分配的，即指定为MISC_DYNAMIC_MINOR，因此，这一步检查就通过了。

函数第17行到第27行为所要注册的日志设备分配从设备号。系统可分配的misc从设备号一共有64个，即从0到63。这些从设备号的使用情况记录在数组misc_minors中，它的定义如下所示。

kernel/goldfish/drivers/char/misc.c

```
1 #define DYNAMIC_MINORS 64 /* like dynamic majors */
2 static unsigned char misc_minors[DYNAMIC_MINORS / 8];
```

数组misc_minors的类型为unsigned char，因此，每一个元素可以管理8个从设备号，64个从设备号就需要8个元素，即数组misc_minors的大小为8。如果某一个从设备号已经被分配了，那么它在数组misc_minors中所对应的位就等于1，否则就等于0。如何在数组misc_minors中找到从设备号i所对应的位呢？首先找到它对应的数组元素位置，即i >> 3，然后取出从设备号i的低三位，即i & 7，那么从设备号i在数组misc_minors中对应的位便是第i >> 3个元素的第i & 7位了。因此，第20行只要检查这一位是否等于0，就可以知道从设备号i是否已经被分配了。如果没有被分配，那么第26行就将它分配给当前正在注册的日志设备，接着第30行在数组misc_minors中记录该从设备号已经被分配。

函数第31行使用misc主设备号MISC_MAJOR和前面得到的从设备号来创建一个设备号对象dev，接着第33行以它为参数调用函数device_create将日志设备misc注册到系统中。MISC_MAJOR是一个

宏，它的定义如下所示。

kernel/goldfish/include/linux/major.h

```
1 #define MISC_MAJOR              10
```

日志设备注册成功之后，函数第44行就将它添加到misc设备列表misc_list中，表示日志设备注册成功了。

第33行调用device_create函数注册日志设备时，最后一个参数用来指定日志设备的名称，即它在设备系统目录/dev中对应的文件名称。从前面的调用过程可以知道，我们要注册的三个日志设备的名称分别为“log_main”、“log_events”和“log_radio”，因此，应当在设备上的/dev目录中看到log_main、log_events和log_radio三个文件。然而，在本章的开头提到，这三个日志设备对应的文件分别为/dev/log/main、/dev/log/events和/dev/log/radio，这是为什么呢？

在前面的2.3.4小节中提到，Android系统使用一种uevent机制来管理系统的设备文件。当Logger日志驱动程序注册一个日志设备时，内核就会发出一个uevent事件，这个uevent最终由init进程中的handle_device_event函数来处理，它的实现如下所示。

system/core/init/devices.c

```
01 static void handle_device_event(struct uevent *uevent)
02 {
03     char devpath[96];
04     int devpath_ready = 0;
05     char *base, *name;
06     char **links = NULL;
07     int block;
08     int i;
09
10     ......
11
12     name = strrchr(uevent->path, '/');
13
14     ......
15
16     name++;
17
18     ......
19
20     if(!strncmp(uevent->subsystem, "block", 5)) {
21         ......
22     } else {
23         ......
24
25         if (!strncmp(uevent->subsystem, "usb", 3)) {
26             ......
27         } else if (!strncmp(uevent->subsystem, "graphics", 8)) {
28             ......
29         } else if (!strncmp(uevent->subsystem, "oncrpc", 6)) {
30             ......
31         } else if (!strncmp(uevent->subsystem, "adsp", 4)) {
32             ......
33         } else if (!strncmp(uevent->subsystem, "msm_camera", 10)) {
34             ......
35         } else if(!strncmp(uevent->subsystem, "input", 5)) {
36             ......
37         } else if(!strncmp(uevent->subsystem, "mtd", 3)) {
38             ......
39         } else if(!strncmp(uevent->subsystem, "sound", 5)) {
40             ......
41         } else if(!strncmp(uevent->subsystem, "misc", 4) &&
42                     !strncmp(name, "log_", 4)) {
```

```
43              base = "/dev/log/";
44              mkdir(base, 0755);
45              name += 4;
46          } else
47              ......
48      }
49
50      if (!devpath_ready)
51          snprintf(devpath, sizeof(devpath), "%s%s", base, name);
52
53      if(!strcmp(uevent->action, "add")) {
54          make_device(devpath, uevent->path, block, uevent->major, uevent->minor);
55          if (links) {
56              for (i = 0; links[i]; i++)
57                  make_link(devpath, links[i]);
58          }
59      }
60
61      ......
62  }
```

函数handle_device_event如果发现新注册的设备类型为misc，并且设备名称是以“log_”开头的，它便会在/dev目录中创建一个log目录，接着将设备名称的前四个字符去掉，即去掉前面的“log_”子字符串，最后就使用新的设备名称在/dev/log目录下创建一个设备文件。因此，我们就不会在设备上的/dev目录中看到log_main、log_events和log_radio三个设备文件，而是在/dev/log目录中看到main、events和radio三个设备文件。

4.2.3　日志设备文件的打开过程

在从Logger日志驱动程序读取或者写入日志记录之前，首先要调用函数open来打开对应的日志设备文件。在Logger日志驱动程序中，日志设备文件的打开函数定义为logger_open，它的实现如下所示。

kernel/goldfish/drivers/staging/android/logger.c

```
01 /*
02  * logger_open - the log's open() file operation
03  *
04  * Note how near a no-op this is in the write-only case. Keep it that way!
05  */
06 static int logger_open(struct inode *inode, struct file *file)
07 {
08          struct logger_log *log;
09          int ret;
10
11          ret = nonseekable_open(inode, file);
12          if (ret)
13                  return ret;
14
15          log = get_log_from_minor(MINOR(inode->i_rdev));
16          if (!log)
17                  return -ENODEV;
18
19          if (file->f_mode & FMODE_READ) {
20                  struct logger_reader *reader;
21
22                  reader = kmalloc(sizeof(struct logger_reader), GFP_KERNEL);
23                  if (!reader)
24                          return -ENOMEM;
25
26                  reader->log = log;
```

```
27                 INIT_LIST_HEAD(&reader->list);
28
29                 mutex_lock(&log->mutex);
30                 reader->r_off = log->head;
31                 list_add_tail(&reader->list, &log->readers);
32                 mutex_unlock(&log->mutex);
33
34                 file->private_data = reader;
35         } else
36                 file->private_data = log;
37
38         return 0;
39 }
```

函数第11行调用函数nonseekable_open将相应的设备文件设置为不可随机访问，即不可以调用函数lseek来设置设备文件的当前读写位置，因为日志记录的读取或者写入都必须按顺序来进行。

函数第15行调用函数get_log_from_minor根据从设备号获得要操作的日志缓冲区结构体。

kernel/goldfish/drivers/staging/android/logger.c

```
01 static struct logger_log * get_log_from_minor(int minor)
02 {
03         if (log_main.misc.minor == minor)
04                 return &log_main;
05         if (log_events.misc.minor == minor)
06                 return &log_events;
07         if (log_radio.misc.minor == minor)
08                 return &log_radio;
09         return NULL;
10 }
```

回到函数logger_open中，对于以写模式打开日志设备的情况，函数logger_open只是把前面获得的日志缓冲区结构体log保存在参数file的成员变量private_data中就结束了；而对于以读模式打开日志设备的情况，函数logger_open的处理就相对复杂一些。函数logger_open首先会为当前进程创建一个logger_reader结构体reader，并且对它进行初始化，其中，最重要的是第30行告诉该进程从log->head位置开始读取日志记录，然后把前面获得的日志缓冲区结构体log保存在结构体reader的成员变量log中，并且把结构体reader保存在参数file的成员变量private_data中。

4.2.4 日志记录的读取过程

当进程调用read函数从日志设备读取日志记录时，Logger日志驱动程序中的函数logger_read就会被调用从相应的日志缓冲区中读取日志记录。

kernel/goldfish/drivers/staging/android/logger.c

```
01 /*
02  * logger_read - our log's read() method
03  *
04  * Behavior:
05  *
06  *      - O_NONBLOCK works
07  *      - If there are no log entries to read, blocks until log is written to
08  *      - Atomically reads exactly one log entry
09  *
10  * Optimal read size is LOGGER_ENTRY_MAX_LEN. Will set errno to EINVAL if read
11  * buffer is insufficient to hold next entry.
12  */
13 static ssize_t logger_read(struct file *file, char __user *buf,
14                            size_t count, loff_t *pos)
15 {
```

```
16          struct logger_reader *reader = file->private_data;
17          struct logger_log *log = reader->log;
18          ssize_t ret;
19          DEFINE_WAIT(wait);
20
21  start:
22          while (1) {
23                  prepare_to_wait(&log->wq, &wait, TASK_INTERRUPTIBLE);
24
25                  mutex_lock(&log->mutex);
26                  ret = (log->w_off == reader->r_off);
27                  mutex_unlock(&log->mutex);
28                  if (!ret)
29                          break;
30
31                  if (file->f_flags & O_NONBLOCK) {
32                          ret = -EAGAIN;
33                          break;
34                  }
35
36                  if (signal_pending(current)) {
37                          ret = -EINTR;
38                          break;
39                  }
40
41                  schedule();
42          }
43
44          finish_wait(&log->wq, &wait);
45          if (ret)
46                  return ret;
47
48          mutex_lock(&log->mutex);
49
50          /* is there still something to read or did we race? */
51          if (unlikely(log->w_off == reader->r_off)) {
52                  mutex_unlock(&log->mutex);
53                  goto start;
54          }
55
56          /* get the size of the next entry */
57          ret = get_entry_len(log, reader->r_off);
58          if (count < ret) {
59                  ret = -EINVAL;
60                  goto out;
61          }
62
63          /* get exactly one entry from the log */
64          ret = do_read_log_to_user(log, reader, buf, ret);
65
66  out:
67          mutex_unlock(&log->mutex);
68
69          return ret;
70  }
```

函数第16行得到日志记录读取进程结构体reader，而第17行得到日志缓冲区结构体log。在前面的4.2.3小节中提到，当进程以读模式打开相应的日志设备时，Logger日志驱动程序就会将上述两个结构体信息保存在打开文件结构体参数file的成员变量private_data中，因此，函数第16行和第17行就可以安全地将它们获取回来。

函数第22行到第42行的while循环用来检查日志缓冲区结构体log中是否有日志记录可读取，这主

要是通过第26行代码来判断的。在进入第22行到第42行的while循环时，第23行首先调用函数prepare_to_wait将等待队列项wait加入日志记录读取进程的等待队列log->wq中。等到确认日志缓冲区结构体log有日志记录可读或者其他原因跳出该while循环时，第44行就会调用函数finish_wait将等待队列项wait从日志记录读取进程的等待队列log->wq中移除。

日志缓冲区结构体log的成员变量w_off用来描述下一条新写入的日志记录在日志缓冲区中的位置，而日志记录读取进程结构体reader的成员变量r_off用来描述当前进程下一条要读取的日志记录在日志缓冲区中的位置。当这两者相等时，就说明日志缓冲区结构体log中没有新的日志记录可读，因此，Logger日志驱动程序就会继续执行第22行到第42行的while循环来等待写入新的日志记录。如果日志缓冲区结构体log中有新的日志记录可读，那么第26行得到的变量ret的值就为false，因此，第29行就会跳出第22行到第42行的while循环，准备读取日志缓冲区结构体log中的日志记录。

如果日志缓冲区结构体log中没有日志记录可读，即第26行得到的变量ret为true，那么当前进程就有可能需要进入睡眠状态，直到日志缓冲区结构体log中有新的日志记录可读为止。当前进程是通过调用第41行的函数schedule来请求内核进行一次进程调度，从而进入睡眠状态的。但是有一种特殊情况——当前进程是以非阻塞模式来打开日志设备文件，即第31行的if语句为true时，当前进程就不会因为日志缓冲区结构体log中没有日志记录可读而进入睡眠状态，它直接返回到用户空间中。一般来说，当驱动程序决定使当前进程进入睡眠状态之前，要先通过调用函数signal_pending来检查当前进程是否有信号正在等待处理。如果有的话，即第36行的if语句为true，那么这时候驱动程序就不能使当前进程进入睡眠状态，而必须要让它结束当前系统调用，以便它可以立即返回去处理那些待处理信号。

如果日志缓冲区结构体log中有日志记录可读，那么Logger日志驱动程序就会跳出第22行到第42行的while循环。但是执行到第51行时，又会再次重新判断日志缓冲区结构体log的成员变量w_off 和日志读取进程结构体reader的成员变量r_off是否相等。这是因为在执行第48行代码来获取互斥量log->mutex时，可能会失败。在这种情况下，当前进程就会进入睡眠状态，直到成功获取互斥量log->mutex为止。在当前进程的睡眠过程中，日志缓冲区结构体log中的日志记录可能已经被其他进程访问过了，即log->w_off的值可能已经发生了变化。因此，第51行需要重新判断日志缓冲区结构体log的成员变量w_off 和日志读取进程结构体reader的成员变量r_off是否相等。

第51行再次确认日志缓冲区结构体log中有新的日志记录可读之后，接着第57行就调用函数get_entry_len来得到下一条要读取的日志记录的长度。

kernel/goldfish/drivers/staging/android/logger.c

```
01 /*
02  * get_entry_len - Grabs the length of the payload of the next entry starting
03  * from 'off'.
04  *
05  * Caller needs to hold log->mutex.
06  */
07 static __u32 get_entry_len(struct logger_log *log, size_t off)
08 {
09         __u16 val;
10
11         switch (log->size - off) {
12         case 1:
13                 memcpy(&val, log->buffer + off, 1);
14                 memcpy(((char *) &val) + 1, log->buffer, 1);
15                 break;
16         default:
17                 memcpy(&val, log->buffer + off, 2);
18         }
19
20         return sizeof(struct logger_entry) + val;
21 }
```

在4.2.1小节中介绍日志记录结构体logger_entry时提到，每一条日志记录都是由两部分内容组成的，其中一部分内容用来描述日志记录，即结构体logger_entry本身所占据的内容；另一部分内容是日志记录的有效负载，即真正的日志记录内容。由于结构体logger_entry的大小是固定的，因此只要知道它的有效负载长度，就可以通过计算得到一条日志记录的长度。日志记录结构体logger_entry的有效负载长度记录在它的成员变量len中。由于成员变量len是日志记录结构体logger_entry的第一个成员变量，因此，它们的起始地址是相同的。又由于日志记录结构体logger_entry的成员变量len的类型为__u16，因此，只要读取该结构体变量地址的前两个字节的内容就可以知道对应的日志记录的长度。我们知道，日志缓冲区是循环使用的，因此，一条日志记录的前两个字节有可能分别保存在日志缓冲区的首尾字节中。因此，在计算一条日志记录的长度时，需要分两种情况来考虑。第一种情况是该日志记录的前两个字节是连在一起的；第二种情况就是前两个字节分别保存在日志缓冲区的首字节和尾字节中。

在函数get_entry_len中，第11行将日志缓冲区结构体log的总长度size减去下一条要读取的日志记录的位置off。如果得到的结果等于1，那么就说明该日志记录的长度值分别保存在日志缓冲区的首尾字节中，因此，第13行和第14行就将它的前后两个字节从日志缓冲区的尾字节和首字节中读取出来，并且将它们的内容组合在变量val中。如果得到的结果不等于1，那么就说明该日志记录的长度值保存在两个连在一起的字节中，即保存在地址log->buffer + off中，因此，第17行就直接将日志记录的长度值读取出来，并且保存在变量val中。最后，第20行将变量val的值加上结构体logger_entry的大小，就得到下一条要读取的日志记录的总长度了。

回到logger_read函数中，第64行调用函数do_read_log_to_user执行真正的日志记录读取操作。

kernel/goldfish/drivers/staging/android/logger.c

```
01 /*
02  * do_read_log_to_user - reads exactly 'count' bytes from 'log' into the
03  * user-space buffer 'buf'. Returns 'count' on success.
04  *
05  * Caller must hold log->mutex.
06  */
07 static ssize_t do_read_log_to_user(struct logger_log *log,
08                                    struct logger_reader *reader,
09                                    char __user *buf,
10                                    size_t count)
11 {
12         size_t len;
13
14         /*
15          * We read from the log in two disjoint operations. First, we read from
16          * the current read head offset up to 'count' bytes or to the end of
17          * the log, whichever comes first.
18          */
19         len = min(count, log->size - reader->r_off);
20         if (copy_to_user(buf, log->buffer + reader->r_off, len))
21                 return -EFAULT;
22
23         /*
24          * Second, we read any remaining bytes, starting back at the head of
25          * the log.
26          */
27         if (count != len)
28                 if (copy_to_user(buf + len, log->buffer, count - len))
29                         return -EFAULT;
30
31         reader->r_off = logger_offset(reader->r_off + count);
32
33         return count;
34 }
```

由于一条日志记录的内容有可能同时位于日志缓冲区的末尾和开头处，即它在地址空间上是不连续的，因此，函数do_read_log_to_user就有可能需要分两次来读取它。第19行计算第一次要读取的日志记录的长度，接着第20行就将这一部分的日志记录复制到用户空间缓冲区buf中。第27行检查上一次是否已经将日志记录的内容读取完毕，如果未读取完毕，即第27行的if语句为true，那么第28行就会继续将第二部分的日志记录复制到用户空间缓冲区buf中，这样就完成了一条日志记录的读取。

注意

日志记录结构体log的成员变量buffer指向的是一个内核缓冲区。在将一个内核缓冲区的内容复制到一个用户空间缓冲区时，必须要调用函数copy_to_user来进行，因为用户空间缓冲区的地址有可能是无效的，而函数copy_to_user在复制内容之前会对它进行检查，避免访问非法地址。

当前进程从日志缓冲区结构体log中读取了一条日志记录之后，就要修改它的成员变量r_off的值，表示下次要读取的是日志缓冲区结构体log中的下一条日志记录。第31行首先将日志读取进程结构体reader的成员变量r_off的值加上前面已经读取的日志记录的长度count，然后再使用宏logger_offset来对计算结果进行调整。

kernel/goldfish/drivers/staging/android/logger.c

```
1 /* logger_offset - returns index 'n' into the log via (optimized) modulus */
2 #define logger_offset(n)        ((n) & (log->size - 1))
```

这是由于日志缓冲区是循环使用的，如果计算得到的下一条日志记录的位置大于日志缓冲区的长度，那么就需要将它绕回到日志缓冲区的前面。

注意

日志缓冲区的长度是2的*N*次方，因此，只要它的值减1之后，再与参数n执行按位与运算，就可以得到参数n在日志缓冲区中的正确位置。

至此，日志记录的读取过程就介绍完了。接下来，我们继续分析日志记录的写入过程。

4.2.5 日志记录的写入过程

当进程调用函数write、writev或者aio_write往日志设备写入日志记录时，Logger日志驱动程序中的函数logger_aio_write就会被调用来执行日志记录的写入操作。

kernel/goldfish/drivers/staging/android/logger.c

```
01 /*
02  * logger_aio_write - our write method, implementing support for write(),
03  * writev(), and aio_write(). Writes are our fast path, and we try to optimize
04  * them above all else.
05  */
06 ssize_t logger_aio_write(struct kiocb *iocb, const struct iovec *iov,
07                          unsigned long nr_segs, loff_t ppos)
08 {
09         struct logger_log *log = file_get_log(iocb->ki_filp);
10         size_t orig = log->w_off;
11         struct logger_entry header;
12         struct timespec now;
13         ssize_t ret = 0;
14
15         now = current_kernel_time();
16
17         header.pid = current->tgid;
```

```
18          header.tid = current->pid;
19          header.sec = now.tv_sec;
20          header.nsec = now.tv_nsec;
21          header.len = min_t(size_t, iocb->ki_left, LOGGER_ENTRY_MAX_PAYLOAD);
22
23          /* null writes succeed, return zero */
24          if (unlikely(!header.len))
25                  return 0;
26
27          mutex_lock(&log->mutex);
28
29          /*
30           * Fix up any readers, pulling them forward to the first readable
31           * entry after (what will be) the new write offset. We do this now
32           * because if we partially fail, we can end up with clobbered log
33           * entries that encroach on readable buffer.
34           */
35          fix_up_readers(log, sizeof(struct logger_entry) + header.len);
36
37          do_write_log(log, &header, sizeof(struct logger_entry));
38
39          while (nr_segs-- > 0) {
40                  size_t len;
41                  ssize_t nr;
42
43                  /* figure out how much of this vector we can keep */
44                  len = min_t(size_t, iov->iov_len, header.len - ret);
45
46                  /* write out this segment's payload */
47                  nr = do_write_log_from_user(log, iov->iov_base, len);
48                  if (unlikely(nr < 0)) {
49                          log->w_off = orig;
50                          mutex_unlock(&log->mutex);
51                          return nr;
52                  }
53
54                  iov++;
55                  ret += nr;
56          }
57
58          mutex_unlock(&log->mutex);
59
60          /* wake up any blocked readers */
61          wake_up_interruptible(&log->wq);
62
63          return ret;
64  }
```

函数的第一个参数iocb表示一个I/O上下文。第二个参数iov保存了要写入的日志记录的内容，它是一个数组向量，长度由第三个参数nr_segs决定。第四个参数ppos指定日志记录的写入位置。由于日志设备中保存了下一条日志记录的写入位置，因此，第四个参数ppos是不需要使用的。

对于类型为main、system和radio的日志记录来说，它们的内容是由优先级（priority）、标签（tag）和内容（msg）三个字段组成的，分别保存在iov[0]、iov[1]和iov[2]中，这时候第三个参数nr_segs的值就为3。对于类型为events的日志记录来说，它们的内容只有标签和内容两个字段，分别保存在iov[0]和iov[1]中，这时候第三个参数nr_segs的值就为2。在特殊情况下，当类型为events的日志记录的内容只有一个值时，第二个参数iov的大小也会等于3，这时候，iov[0]表示日志标签，iov[1]表示日志记录的内容值类型，iov[2]表示日志记录的内容值。在后面的4.4小节中分析日志的写入接口时，我们再详细介绍参数iov的使用方法。无论要写入的是什么类型的日志记录，它的总长度值都是保存在

参数iocb的成员变量ki_left中的，单位是字节。

在前面的4.2.3小节提到，当进程以写模式打开日志设备时，Logger日志驱动程序会把对应的日志缓冲区结构体log保存在一个打开文件结构体file的成员变量private_data中。参数iocb的成员变量ki_filp正好指向该打开文件结构体，因此，第9行就调用函数file_get_log安全地将对应的日志缓冲区结构体log获取回来。

kernel/goldfish/drivers/staging/android/logger.c

```
1 static inline struct logger_log * file_get_log(struct file *file)
2 {
3         if (file->f_mode & FMODE_READ) {
4                 struct logger_reader *reader = file->private_data;
5                 return reader->log;
6         } else
7                 return file->private_data;
8 }
```

有了这个日志缓冲区结构体log之后，就可以执行日志记录的写入操作了。

回到函数logger_aio_write中，第11行首先定义一个日志记录结构体header，接着第17行到第21行对它进行初始化，即记录日志记录写入进程的TGID和PID，以及日志记录的写入时间和长度。由于一条日志记录的最大长度为LOGGER_ENTRY_MAX_PAYLOAD，因此，第21行在初始化日志记录的长度时，要取参数iocb的成员变量ki_left和宏LOGGER_ENTRY_MAX_PAYLOAD的较小值。

在将日志记录写入到日志缓冲区之前，还有一件重要的事情要做，就是修正那些正在读取该日志缓冲区的进程的当前日志记录的读取位置，即日志读取进程结构体logger_reader的成员变量r_off的值，以及修正新的日志读取进程的日志记录开始读取位置，即日志缓冲区结构体logger_log的成员变量head的值。我们知道，日志缓冲区是循环使用的，即当日志缓冲区写满之后，新的日志记录会覆盖旧的日志记录，而这些将要被覆盖的日志记录有可能正好是某些进程的下一条日志记录的读取位置。由于这些位置即将被新的日志记录覆盖，因此，我们就需要对它们进行修正，以便这些受影响的进程下次能够读取到正确的日志记录。这个修正工作要在日志记录写入前去做，这是为了保证在日志记录写入的过程中出现部分失败时，不会破坏整个日志缓冲区的内容。

修正日志记录读取进程的读取位置的工作是通过调用函数fix_up_readers来完成的，它的实现如下所示。

kernel/goldfish/drivers/staging/android/logger.c

```
01 /*
02  * fix_up_readers - walk the list of all readers and "fix up" any who were
03  * lapped by the writer; also do the same for the default "start head".
04  * We do this by "pulling forward" the readers and start head to the first
05  * entry after the new write head.
06  *
07  * The caller needs to hold log->mutex.
08  */
09 static void fix_up_readers(struct logger_log *log, size_t len)
10 {
11         size_t old = log->w_off;
12         size_t new = logger_offset(old + len);
13         struct logger_reader *reader;
14
15         if (clock_interval(old, new, log->head))
16                 log->head = get_next_entry(log, log->head, len);
17
18         list_for_each_entry(reader, &log->readers, list)
19                 if (clock_interval(old, new, reader->r_off))
20                         reader->r_off = get_next_entry(log, reader->r_off, len);
21 }
```

第二个参数len表示即将要写入的日志记录的总长度，它等于日志记录结构体logger_entry的长度加上日志记录的有效负载长度。第11行得到日志缓冲区结构体log的下一条日志记录的写入位置，保存在变量old中。第12行计算将新的日志记录写入到日志缓冲区结构体log之后，下一条日志记录的写入位置，保存在变量new中。位于old值和new值之间的那些日志记录读取位置是无效的，因为它们即将被新写入的日志记录覆盖。

给定一个日志记录读取位置c，判断它是否位于a和b之间是通过调用函数clock_interval来实现的，如下所示。

kernel/goldfish/drivers/staging/android/logger.c

```
01 /*
02  * clock_interval - is a < c < b in mod-space? Put another way, does the line
03  * from a to b cross c?
04  */
05 static inline int clock_interval(size_t a, size_t b, size_t c)
06 {
07         if (b < a) {
08                 if (a < c || b >= c)
09                         return 1;
10         } else {
11                 if (a < c && b >= c)
12                         return 1;
13         }
14
15         return 0;
16 }
```

函数clock_interval分两种情况来判断c是否位于a和b之间：第一种情况是b小于a；第二种情况是b大于等于a。在这两种情况下，如果c位于a和b之间，那么函数的返回值为1，否则为0。

回到函数fix_up_readers中，第16行修正的是新的日志读取进程在日志缓冲区结构体log中的开始读取位置，即日志缓冲区结构体log的成员变量head的值。第18行到第20行的循环语句修正的是那些正在读取日志缓冲区结构体log中的日志记录的进程的下一条日志记录的位置，即日志读取进程结构体reader的成员变量r_off的值。如果这些位置正好位于变量old和new的值之间，那么就调用函数get_next_entry将它们的值修改为下一条日志记录的位置。

kernel/goldfish/drivers/staging/android/logger.c

```
01 /*
02  * get_next_entry - return the offset of the first valid entry at least 'len'
03  * bytes after 'off'.
04  *
05  * Caller must hold log->mutex.
06  */
07 static size_t get_next_entry(struct logger_log *log, size_t off, size_t len)
08 {
09         size_t count = 0;
10
11         do {
12                 size_t nr = get_entry_len(log, off);
13                 off = logger_offset(off + nr);
14                 count += nr;
15         } while (count < len);
16
17         return off;
18 }
```

参数off表示要修正的位置，而参数len表示即将要写入的日志记录的长度。第11行到第15行的while循环首先调用函数get_entry_len来获得在位置off的日志记录的长度，然后将它增加到位置off中去。如果累计增加的位置大于或者等于参数len的值，那么就完成了对位置off的修正；否则，就要继

续将位置off移到再下一条日志记录的位置。

回到函数logger_aio_write中，第37行到第56行代码用来将参数iov中的日志记录内容写入到日志缓冲区结构体log中。由于一条日志记录在日志缓冲区中是由一个日志记录结构体和一个有效负载组成的，因此，函数logger_aio_write就分两步将它们写入到日志缓冲区中。

函数前面已经为即将要写入的日志记录准备好了一个日志记录结构体header，因此，第37行就调用函数do_write_log将它的内容写入到日志缓冲区结构体log中。

kernel/goldfish/drivers/staging/android/logger.c

```
01 /*
02  * do_write_log - writes 'len' bytes from 'buf' to 'log'
03  *
04  * The caller needs to hold log->mutex.
05  */
06 static void do_write_log(struct logger_log *log, const void *buf, size_t count)
07 {
08         size_t len;
09
10         len = min(count, log->size - log->w_off);
11         memcpy(log->buffer + log->w_off, buf, len);
12
13         if (count != len)
14                 memcpy(log->buffer, buf + len, count - len);
15
16         log->w_off = logger_offset(log->w_off + count);
17
18 }
```

由于一个日志记录结构体的内容可能会被分别写在日志缓冲区的尾部和头部，因此，函数do_write_log就分两次来写入它的内容。第10行计算写入到日志缓冲区尾部的内容的长度，接着第11行就直接将对应的日志记录结构体的内容复制到日志缓冲区中。第13行判断前面是否已经将日志记录结构体的内容全部写入到日志缓冲区中了。如果不是，第14行就会将剩余的内容写入到日志缓冲区的头部。

注意

参数buf指向的内容即为要写入的日志记录结构体的内容，它位于内核空间中，因此，第11行和第14行将它的内容写入到日志缓冲区中时，直接调用函数memcpy来复制就可以了。将日志记录结构体的内容写入到日志缓冲区中之后，第16行就要更新它的成员变量w_off的值，因为它始终指向下一条日志记录的写入位置。

回到函数logger_aio_write中，第39行到第56行的while循环把参数iov的内容写入到日志缓冲区结构体log中。参数iov里面所保存的内容对应于要写入的日志记录的有效负载，它们是从用户空间传进来的，因此，函数logger_aio_write不能直接调用函数memcpy将它们复制到日志缓冲区结构体log中，而是调用函数do_write_log_from_user来执行复制操作。

kernel/goldfish/drivers/staging/android/logger.c

```
01 /*
02  * do_write_log_user - writes 'len' bytes from the user-space buffer 'buf' to
03  * the log 'log'
04  *
05  * The caller needs to hold log->mutex.
06  *
07  * Returns 'count' on success, negative error code on failure.
08  */
09 static ssize_t do_write_log_from_user(struct logger_log *log,
10                                       const void __user *buf, size_t count)
```

```
11 {
12         size_t len;
13
14         len = min(count, log->size - log->w_off);
15         if (len && copy_from_user(log->buffer + log->w_off, buf, len))
16                 return -EFAULT;
17
18         if (count != len)
19                 if (copy_from_user(log->buffer, buf + len, count - len))
20                         return -EFAULT;
21
22         log->w_off = logger_offset(log->w_off + count);
23
24         return count;
25 }
```

与日志记录结构体的写入过程类似，日志记录的有效负载也有可能会被分别写入到日志缓冲区的尾部和头部，因此，函数do_write_log_from_user就分两次来写入它的内容。第14行计算写入到日志缓冲区尾部的内容的长度，接着第15行就调用函数copy_from_user将对应的日志记录的有效负载复制到日志缓冲区中。第18行判断前面是否已经将日志记录的有效负载全部写入到日志缓冲区了。如果不是，第19行就会将剩余的内容写入到日志缓冲区的头部。

注意

参数buf指向的内容是保存在用户空间中的，因此，第15行和第19行将它的内容写入到日志缓冲区中时，需要调用函数copy_from_user来执行复制操作。将日志记录的有效负载写入到日志缓冲区中之后，接下来第22行就要更新它的成员变量w_off的值，因为它始终指向下一条日志记录的写入位置。

在前面的4.2.4小节中，我们在分析日志记录的读取过程时提到，如果日志缓冲区当前没有新的日志记录可读，那么日志读取进程就会进入到日志缓冲区的等待队列wq中去睡眠等写入新的日志记录。现在既然已经往日志缓冲区中写入新的日志记录了，函数logger_aio_write在第61行就会调用函数wake_up_interruptible来唤醒那些睡眠在日志缓冲区的等待队列wq中的进程，目的是通知它们有新的日志记录可读了。

4.3　运行时库层日志库

Android系统在运行时库层提供了一个用来和Logger日志驱动程序进行交互的日志库liblog。通过日志库liblog提供的接口，应用程序就可以方便地往Logger日志驱动程序中写入日志记录。位于运行时库层的C/C++日志写入接口和位于应用程序框架层的Java日志写入接口都是通过liblog库提供的日志写入接口来往Logger日志驱动程序中写入日志记录的，因此，在分析这些C/C++或者Java日志写入接口之前，我们首先介绍liblog库的日志记录写入接口。

日志库liblog提供的日志记录写入接口实现在logd_write.c文件中，它的位置如下：

```
~/Android/system/core
----liblog
     ----logd_write.c
```

它里面实现了一系列的日志记录写入函数，如图4-7所示。

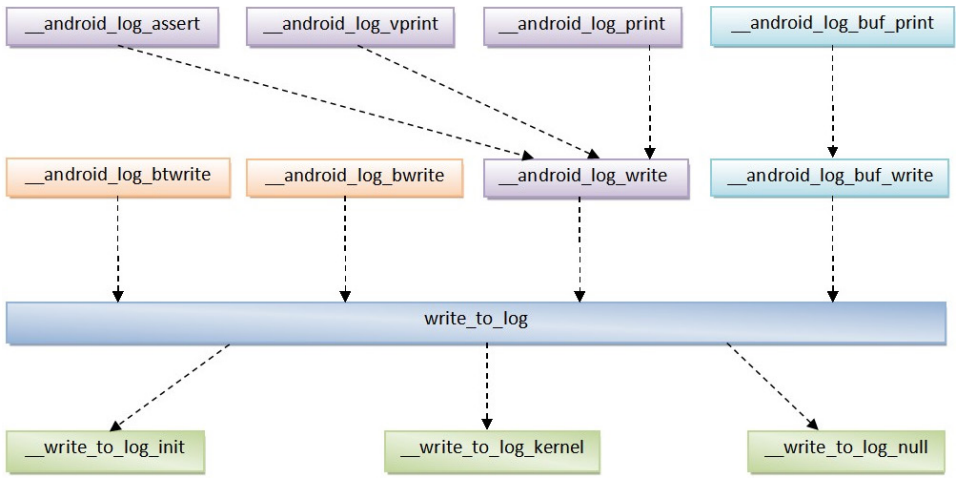


图4-7　liblog日志库的函数调用关系

根据写入的日志记录的类型不同，这些函数可以划分为三个类别。其中，函数__android_log_assert、__android_log_vprint和__android_log_print用来写入类型为main的日志记录；函数__android_log_btwrite和__android_log_bwrite用来写入类型为events的日志记录；函数__android_log_buf_print可以写入任意一种类型的日志记录。特别地，在函数__android_log_write和__android_log_buf_write中，如果要写入的日志记录的标签以“RIL”开头或者等于“HTC_RIL”、“AT”、“GSM”、“STK”、“CDMA”、“PHONE”或“SMS”，那么它们就会被认为是radio类型的日志记录。

无论写入的是什么类型的日志记录，它们最终都是通过调用函数write_to_log写入到Logger日志驱动程序中的。write_to_log是一个函数指针，它开始时指向函数__write_to_log_init。因此，当函数write_to_log第一次被调用时，实际上执行的是函数__write_to_log_init。函数__write_to_log_init执行的是一些日志库的初始化操作，接着将函数指针write_to_log重定向到函数__write_to_log_kernel或者__write_to_log_null中，这取决于是否成功地将日志设备文件打开。

在本节接下来的内容中，我们就分别描述日志库liblog提供的日志记录写入函数的实现。

write_to_log

system/core/liblog/logd_write.c

```
1 static int __write_to_log_init(log_id_t, struct iovec *vec, size_t nr);
2 static int (*write_to_log)(log_id_t, struct iovec *vec, size_t nr) = __write_to_log_init;
```

函数指针write_to_log在开始的时候被设置为函数__write_to_log_init。当它第一次被调用时，便会执行函数__write_to_log_init来初始化日志库liblog，如下所示。

system/core/liblog/logd_write.c

```
01 static int log_fds[(int)LOG_ID_MAX] = { -1, -1, -1, -1 };
02
03 static int __write_to_log_init(log_id_t log_id, struct iovec *vec, size_t nr)
04 {
05     ......
06
07     if (write_to_log == __write_to_log_init) {
08         log_fds[LOG_ID_MAIN] = log_open("/dev/"LOGGER_LOG_MAIN, O_WRONLY);
09         log_fds[LOG_ID_RADIO] = log_open("/dev/"LOGGER_LOG_RADIO, O_WRONLY);
10         log_fds[LOG_ID_EVENTS] = log_open("/dev/"LOGGER_LOG_EVENTS, O_WRONLY);
11         log_fds[LOG_ID_SYSTEM] = log_open("/dev/"LOGGER_LOG_SYSTEM, O_WRONLY);
12
13         write_to_log = __write_to_log_kernel;
14
```

```
15          if (log_fds[LOG_ID_MAIN] < 0 || log_fds[LOG_ID_RADIO] < 0 ||
16                  log_fds[LOG_ID_EVENTS] < 0) {
17              log_close(log_fds[LOG_ID_MAIN]);
18              log_close(log_fds[LOG_ID_RADIO]);
19              log_close(log_fds[LOG_ID_EVENTS]);
20              log_fds[LOG_ID_MAIN] = -1;
21              log_fds[LOG_ID_RADIO] = -1;
22              log_fds[LOG_ID_EVENTS] = -1;
23              write_to_log = __write_to_log_null;
24          }
25
26          if (log_fds[LOG_ID_SYSTEM] < 0) {
27              log_fds[LOG_ID_SYSTEM] = log_fds[LOG_ID_MAIN];
28          }
29      }
30
31      ......
32
33      return write_to_log(log_id, vec, nr);
34  }
```

在函数__write_to_log_init中，第7行如果发现函数指针write_to_log指向的是自己，那么就会调用函数open打开系统中的日志设备文件，并且把得到的文件描述符保存在全局数组log_fds中。

LOG_ID_MAIN、LOG_ID_RADIO、LOG_ID_EVENTS、LOG_ID_SYSTEM和LOG_ID_MAX是五个枚举值，它们的定义如下所示。

system/core/include/cutils/log.h

```
1 typedef enum {
2     LOG_ID_MAIN = 0,
3     LOG_ID_RADIO = 1,
4     LOG_ID_EVENTS = 2,
5     LOG_ID_SYSTEM = 3,
6
7     LOG_ID_MAX
8 } log_id_t;
```

LOGGER_LOG_MAIN、LOGGER_LOG_RADIO、LOGGER_LOG_EVENTS和LOGGER_LOG_SYSTEM是四个宏，它们的定义如下所示。

system/core/include/cutils/logger.h

```
1 #define LOGGER_LOG_MAIN         "log/main"
2 #define LOGGER_LOG_RADIO        "log/radio"
3 #define LOGGER_LOG_EVENTS       "log/events"
4 #define LOGGER_LOG_SYSTEM       "log/system"
```

因此，函数__write_to_log_init的第8行到第11行实际上是调用宏log_open来打开/dev/log/main、/dev/log/radio、/dev/log/events和/dev/log/system四个日志设备文件。宏log_open的定义如下所示。

system/core/liblog/logd_write.c

```
01 #if FAKE_LOG_DEVICE
02 // This will be defined when building for the host.
03 #define log_open(pathname, flags) fakeLogOpen(pathname, flags)
04 #define log_writev(filedes, vector, count) fakeLogWritev(filedes, vector, count)
05 #define log_close(filedes) fakeLogClose(filedes)
06 #else
07 #define log_open(pathname, flags) open(pathname, flags)
08 #define log_writev(filedes, vector, count) writev(filedes, vector, count)
09 #define log_close(filedes) close(filedes)
10 #endif
```

在正式环境中编译日志库liblog时，宏FAKE_LOG_DEVICE的值定义为0，因此，宏log_open实际

上指向的是打开文件操作函数open。从这里同时也可以看到，在正式环境中，宏log_writev和log_close分别指向写文件操作函数writev和关闭文件操作函数close。

回到函数__write_to_log_init中，第15行的if语句判断/dev/log/main、/dev/log/radio和/dev/log/events三个日志设备文件是否都打开成功。如果是，就将函数指针write_to_log指向函数__write_to_log_kernel；否则，将函数指针write_to_log指向函数__write_to_log_null。第26行的if语句判断日志设备文件/dev/log/system是否打开成功。如果不成功，就将log_fds[LOG_ID_SYSTEM]的值设置为log_fds[LOG_ID_MAIN]，即将类型为system和main的日志记录都写入到日志设备文件/dev/log/main中。

__write_to_log_kernel

system/core/liblog/logd_write.c

```
01 static int __write_to_log_kernel(log_id_t log_id, struct iovec *vec, size_t nr)
02 {
03     ssize_t ret;
04     int log_fd;
05
06     if (/*(int)log_id >= 0 &&*/ (int)log_id < (int)LOG_ID_MAX) {
07         log_fd = log_fds[(int)log_id];
08     } else {
09         return EBADF;
10     }
11
12     do {
13         ret = log_writev(log_fd, vec, nr);
14     } while (ret < 0 && errno == EINTR);
15
16     return ret;
17 }
```

函数__write_to_log_kernel根据参数log_id在全局数组log_fds中找到对应的日志设备文件描述符，然后调用宏log_writev，即函数writev，把日志记录写入到Logger日志驱动程序中。

注意

如果调用宏log_writev写入日志记录时，Logger日志驱动程序的返回值小于0，并且错误码等于EINTR，那么就需要重新执行写入日志记录的操作。这种情况一般出现在当前进程等待写入日志记录的过程中，刚好碰到有新的信号需要处理，这时候内核就会返回一个EINTR错误码给调用者，表示需要调用者再次执行相同的操作。

__write_to_log_null

system/core/liblog/logd_write.c

```
1 static int __write_to_log_null(log_id_t log_fd, struct iovec *vec, size_t nr)
2 {
3     return -1;
4 }
```

函数__write_to_log_null是一个空实现，什么也不做。在日志设备文件打开失败的情况下，函数指针write_to_log才会指向该函数。

__android_log_write

system/core/liblog/logd_write.c

```
01 int __android_log_write(int prio, const char *tag, const char *msg)
02 {
03     struct iovec vec[3];
04     log_id_t log_id = LOG_ID_MAIN;
```

```
05
06      if (!tag)
07          tag = "";
08
09      /* XXX: This needs to go! */
10      if (!strcmp(tag, "HTC_RIL") ||
11          !strncmp(tag, "RIL", 3) || /* Any log tag with "RIL" as the prefix */
12          !strcmp(tag, "AT") ||
13          !strcmp(tag, "GSM") ||
14          !strcmp(tag, "STK") ||
15          !strcmp(tag, "CDMA") ||
16          !strcmp(tag, "PHONE") ||
17          !strcmp(tag, "SMS"))
18              log_id = LOG_ID_RADIO;
19
20      vec[0].iov_base   = (unsigned char *) &prio;
21      vec[0].iov_len    = 1;
22      vec[1].iov_base   = (void *) tag;
23      vec[1].iov_len    = strlen(tag) + 1;
24      vec[2].iov_base   = (void *) msg;
25      vec[2].iov_len    = strlen(msg) + 1;
26
27      return write_to_log(log_id, vec, 3);
28  }
```

在默认情况下，函数__android_log_write写入的日志记录的类型为main。然而，如果传进来的日志记录的标签以“RIL”开头或者等于“HTC_RIL”、“AT”、“GSM”、“STK”、“CDMA”、“PHONE”或“SMS”，那么它就会被认为是类型为radio的日志记录。

注意

第20行到第25行首先将日志记录的优先级、标签和内容保存在数组元素vec[0]、vec[1]和vec[2]中，然后再将它们写入到Logger日志驱动程序中。日志记录的标签和内容的类型均为字符串，它们后面紧跟着的字符串结束字符‘\0’也被写入到Logger日志驱动程序中。这样做的好处是，可以通过字符串结束字符‘\0’来解析日志记录的标签字段和内容字段。

__android_log_buf_write

system/core/liblog/logd_write.c

```
01 int __android_log_buf_write(int bufID, int prio, const char *tag, const char *msg)
02 {
03      struct iovec vec[3];
04
05      if (!tag)
06          tag = "";
07
08      /* XXX: This needs to go! */
09      if (!strcmp(tag, "HTC_RIL") ||
10          !strncmp(tag, "RIL", 3) || /* Any log tag with "RIL" as the prefix */
11          !strcmp(tag, "AT") ||
12          !strcmp(tag, "GSM") ||
13          !strcmp(tag, "STK") ||
14          !strcmp(tag, "CDMA") ||
15          !strcmp(tag, "PHONE") ||
16          !strcmp(tag, "SMS"))
17              bufID = LOG_ID_RADIO;
18
19      vec[0].iov_base   = (unsigned char *) &prio;
20      vec[0].iov_len    = 1;
21      vec[1].iov_base   = (void *) tag;
```

```
22      vec[1].iov_len    = strlen(tag) + 1;
23      vec[2].iov_base   = (void *) msg;
24      vec[2].iov_len    = strlen(msg) + 1;
25
26      return write_to_log(bufID, vec, 3);
27  }
```

函数__android_log_buf_write的实现与函数__android_log_write的实现类似，不过它可以指定写入的日志记录的类型。特别地，如果要写入的日志记录的标签以“RIL”开头或者等于“HTC_RIL”、“AT”、“GSM”、“STK”、“CDMA”、“PHONE”或“SMS”，那么它们就会被认为是类型为radio的日志记录。

函数__android_log_buf_write与前面分析的函数__android_log_write一样，把紧跟在日志记录标签和内容后面的字符串结束符号‘\0’也写入到Logger日志驱动程序中，目的也是为了以后从Logger日志驱动程序读取日志时，可以方便地将日志记录的标签字段和内容字段解析出来。

__android_log_vprint、__android_log_print、__android_log_assert

system/core/liblog/logd_write.c

```
01 int __android_log_vprint(int prio, const char *tag, const char *fmt, va_list ap)
02 {
03     char buf[LOG_BUF_SIZE];
04
05     vsnprintf(buf, LOG_BUF_SIZE, fmt, ap);
06
07     return __android_log_write(prio, tag, buf);
08 }
09
10 int __android_log_print(int prio, const char *tag, const char *fmt, ...)
11 {
12     va_list ap;
13     char buf[LOG_BUF_SIZE];
14
15     va_start(ap, fmt);
16     vsnprintf(buf, LOG_BUF_SIZE, fmt, ap);
17     va_end(ap);
18
19     return __android_log_write(prio, tag, buf);
20 }
21
22 void __android_log_assert(const char *cond, const char *tag,
23                           const char *fmt, ...)
24 {
25     char buf[LOG_BUF_SIZE];
26
27     if (fmt) {
28         va_list ap;
29         va_start(ap, fmt);
30         vsnprintf(buf, LOG_BUF_SIZE, fmt, ap);
31         va_end(ap);
32     } else {
33         /* Msg not provided, log condition.  N.B. Do not use cond directly as
34          * format string as it could contain spurious '%' syntax (e.g.
35          * "%d" in "blocks%devs == 0").
36          */
37         if (cond)
38             snprintf(buf, LOG_BUF_SIZE, "Assertion failed: %s", cond);
39         else
40             strcpy(buf, "Unspecified assertion failed");
41     }
42
```

```
43      __android_log_write(ANDROID_LOG_FATAL, tag, buf);
44
45      __builtin_trap(); /* trap so we have a chance to debug the situation */
46  }
```

函数__android_log_vprint、__android_log_print和__android_log_assert都是调用函数__android_log_write向Logger日志驱动程序中写入日志记录的，它们都可以使用格式化字符串来描述要写入的日志记录内容。

__android_log_buf_print

system/core/liblog/logd_write.c

```
01 int __android_log_buf_print(int bufID, int prio, const char *tag, const char *fmt, ...)
02 {
03      va_list ap;
04      char buf[LOG_BUF_SIZE];
05
06      va_start(ap, fmt);
07      vsnprintf(buf, LOG_BUF_SIZE, fmt, ap);
08      va_end(ap);
09
10      return __android_log_buf_write(bufID, prio, tag, buf);
11  }
```

函数__android_log_buf_print是调用函数__android_log_buf_write向Logger日志驱动程序中写入日志记录的，它可以指定要写入的日志记录的类型，以及使用格式化字符串来描述要写入的日志记录内容。

__android_log_bwrite、__android_log_btwrite

system/core/liblog/logd_write.c

```
01 int __android_log_bwrite(int32_t tag, const void *payload, size_t len)
02 {
03      struct iovec vec[2];
04
05      vec[0].iov_base = &tag;
06      vec[0].iov_len = sizeof(tag);
07      vec[1].iov_base = (void*)payload;
08      vec[1].iov_len = len;
09
10      return write_to_log(LOG_ID_EVENTS, vec, 2);
11  }
12
13  /*
14   * Like __android_log_bwrite, but takes the type as well.  Doesn't work
15   * for the general case where we're generating lists of stuff, but very
16   * handy if we just want to dump an integer into the log.
17   */
18  int __android_log_btwrite(int32_t tag, char type, const void *payload,
19      size_t len)
20  {
21      struct iovec vec[3];
22
23      vec[0].iov_base = &tag;
24      vec[0].iov_len = sizeof(tag);
25      vec[1].iov_base = &type;
26      vec[1].iov_len = sizeof(type);
27      vec[2].iov_base = (void*)payload;
28      vec[2].iov_len = len;
29
30      return write_to_log(LOG_ID_EVENTS, vec, 3);
31  }
```

函数__android_log_bwrite和__android_log_btwrite写入的日志记录的类型为events。其中，函数__android_log_bwrite写入的日志记录的内容可以由多个值组成，而函数__android_log_btwrite写入的日志记录的内容只有一个值。在前面的4.1小节中提到，类型为events的日志记录的内容一般是由一系列值组成的，每一个值都有自己的名称、类型和单位。函数__android_log_btwrite就是通过第二个参数type来指定要写入的日志记录内容的值类型的，由于它写入的日志记录的内容只有一个值，因此，为了方便读取，就把这个值的类型抽取出来，作为一个独立的字段写入到Logger日志驱动程序中。

4.4 C/C++日志写入接口

在前面的4.3小节中，我们介绍了Android系统运行时库层的日志库写入接口，但是在实际开发中，我们一般不会直接使用这些接口来写日志。在实际开发中，我们常常希望有些日志只在程序的调试版本中输出，而不希望它们在发布版本中输出。Android系统就提供了三组常用的C/C++宏来封装日志写入接口，这些宏有的在程序的非调试版本中只是一个空定义，因此，可以避免在程序的发布版本中输出日志。

第一组宏是LOGV、LOGD、LOGI、LOGW和LOGE，它们用来写入类型为main的日志记录；第二组宏是SLOGV、SLOGD、SLOGI、SLOGW和SLOGE，它们用来写入类型为system的日志记录；第三组宏是LOG_EVENT_INT、LOG_EVENT_LONG和LOG_EVENT_STRING，它们用来写入类型为events的日志记录。这些宏定义在Android系统运行时库层的一个头文件log.h中，它的位置如下所示。

```
~/Android/system/core/include
----cutils
     ----log.h
```

这个头文件定义了一个宏LOG_NDEBUG，用来区分程序是调试版本还是发布版本，如下所示。

system/core/include/cutils/log.h

```
01 /*
02  * Normally we strip LOGV (VERBOSE messages) from release builds.
03  * You can modify this (for example with "#define LOG_NDEBUG 0"
04  * at the top of your source file) to change that behavior.
05  */
06 #ifndef LOG_NDEBUG
07 #ifdef NDEBUG
08 #define LOG_NDEBUG 1
09 #else
10 #define LOG_NDEBUG 0
11 #endif
12 #endif
```

在程序的发布版本中，宏LOG_NDEBUG定义为1，而在调试版本中定义为0。通过这个宏，我们就可以将某些日志宏在程序的发布版本中定义为空，从而限制它们在程序的发布版本中输出。

这个头文件还定义了宏LOG_TAG，用作当前编译单元的默认日志记录标签，它的定义如下所示。

system/core/include/cutils/log.h

```
1 /*
2  * This is the local tag used for the following simplified
3  * logging macros.  You can change this preprocessor definition
4  * before using the other macros to change the tag.
5  */
6 #ifndef LOG_TAG
7 #define LOG_TAG NULL
8 #endif
```

它默认定义为NULL，即没有日志记录标签。如果一个模块想要定义自己的默认日志记录标签，那么就需要使用#define指令来自定义宏LOG_TAG的值。

了解了这两个宏的定义之后，我们就开始分析这三组C/C++日志宏的实现。

LOGV、LOGD、LOGI、LOGW和LOGE

system/core/include/cutils/log.h

```
01 /*
02  * Simplified macro to send a verbose log message using the current LOG_TAG.
03  */
04 #ifndef LOGV
05 #if LOG_NDEBUG
06 #define LOGV(...)   ((void)0)
07 #else
08 #define LOGV(...) ((void)LOG(LOG_VERBOSE, LOG_TAG, __VA_ARGS__))
09 #endif
10 #endif
11
12 /*
13  * Simplified macro to send a debug log message using the current LOG_TAG.
14  */
15 #ifndef LOGD
16 #define LOGD(...) ((void)LOG(LOG_DEBUG, LOG_TAG, __VA_ARGS__))
17 #endif
18
19 /*
20  * Simplified macro to send an info log message using the current LOG_TAG.
21  */
22 #ifndef LOGI
23 #define LOGI(...) ((void)LOG(LOG_INFO, LOG_TAG, __VA_ARGS__))
24 #endif
25
26 /*
27  * Simplified macro to send a warning log message using the current LOG_TAG.
28  */
29 #ifndef LOGW
30 #define LOGW(...) ((void)LOG(LOG_WARN, LOG_TAG, __VA_ARGS__))
31 #endif
32
33 /*
34  * Simplified macro to send an error log message using the current LOG_TAG.
35  */
36 #ifndef LOGE
37 #define LOGE(...) ((void)LOG(LOG_ERROR, LOG_TAG, __VA_ARGS__))
38 #endif
```

这五个宏是用来写入类型为main的日志记录的，它们写入的日志记录的优先级分别为VERBOSE、DEBUG、INFO、WARN和ERROR。其中，宏LOGV只有在宏LOG_NDEBUG定义为0时，即在程序的调试版本中，才是有效的；否则，它只是一个空定义。

这五个宏是通过使用宏LOG来实现日志写入功能的，它的定义如下所示。

system/core/include/cutils/log.h

```
01 /*
02  * Basic log message macro.
03  *
04  * Example:
05  *  LOG(LOG_WARN, NULL, "Failed with error %d", errno);
06  *
07  * The second argument may be NULL or "" to indicate the "global" tag.
08  */
09 #ifndef LOG
```

```
10 #define LOG(priority, tag, ...) \
11     LOG_PRI(ANDROID_##priority, tag, __VA_ARGS__)
12 #endif
13
14 /*
15  * Log macro that allows you to specify a number for the priority.
16  */
17 #ifndef LOG_PRI
18 #define LOG_PRI(priority, tag, ...) \
19     android_printLog(priority, tag, __VA_ARGS__)
20 #endif
21
22 #define android_printLog(prio, tag, fmt...) \
23     __android_log_print(prio, tag, fmt)
```

当宏LOG展开后，它的第一个参数priority加上前缀“ANDROID_”之后，就变成了另外一个宏LOG_PRI的第一个参数。例如，宏LOGV展开后就得到宏LOG_PRI的第一个参数为ANDROID_LOG_VERBOSE。这些形式为ANDROID_##priority的参数都是类型为android_LogPriority的枚举值，它们的定义如下所示。

system/core/include/android/log.h

```
01 /*
02  * Android log priority values, in ascending priority order.
03  */
04 typedef enum android_LogPriority {
05     ANDROID_LOG_UNKNOWN = 0,
06     ANDROID_LOG_DEFAULT,    /* only for SetMinPriority() */
07     ANDROID_LOG_VERBOSE,
08     ANDROID_LOG_DEBUG,
09     ANDROID_LOG_INFO,
10     ANDROID_LOG_WARN,
11     ANDROID_LOG_ERROR,
12     ANDROID_LOG_FATAL,
13     ANDROID_LOG_SILENT,     /* only for SetMinPriority(); must be last */
14 } android_LogPriority;
```

回到宏LOG_PRI的定义中，它最终是通过调用日志库liblog提供的函数__android_log_print向Logger日志驱动程序中写入日志记录的。函数__android_log_print的实现可以参考前面4.3小节的内容，这里不再详述。

SLOGV、SLOGD、SLOGI、SLOGW和SLOGE

system/core/include/cutils/log.h

```
01 /*
02  * Simplified macro to send a verbose system log message using the current LOG_TAG.
03  */
04 #ifndef SLOGV
05 #if LOG_NDEBUG
06 #define SLOGV(...)   ((void)0)
07 #else
08 #define SLOGV(...) ((void)__android_log_buf_print(LOG_ID_SYSTEM, ANDROID_LOG_VERBOSE, LOG_TAG, __VA_ARGS__))
09 #endif
10 #endif
11
12 /*
13  * Simplified macro to send a debug system log message using the current LOG_TAG.
14  */
15 #ifndef SLOGD
16 #define SLOGD(...) ((void)__android_log_buf_print(LOG_ID_SYSTEM, ANDROID_LOG_DEBUG, LOG_TAG, __VA_ARGS__))
17 #endif
18
19 /*
20  * Simplified macro to send an info system log message using the current LOG_TAG.
```

```
 */
#ifndef SLOGI
#define SLOGI(...) ((void)__android_log_buf_print(LOG_ID_SYSTEM, ANDROID_LOG_INFO, LOG_TAG, __VA_ARGS__))
#endif

/*
 * Simplified macro to send a warning system log message using the current LOG_TAG.
 */
#ifndef SLOGW
#define SLOGW(...) ((void)__android_log_buf_print(LOG_ID_SYSTEM, ANDROID_LOG_WARN, LOG_TAG, __VA_ARGS__))
#endif

/*
 * Simplified macro to send an error system log message using the current LOG_TAG.
 */
#ifndef SLOGE
#define SLOGE(...) ((void)__android_log_buf_print(LOG_ID_SYSTEM, ANDROID_LOG_ERROR, LOG_TAG, __VA_ARGS__))
#endif
```

这五个宏是用来写入类型为system的日志记录的，它们写入的日志记录的优先级分别为VERBOSE、DEBUG、INFO、WARN和ERROR。其中，宏SLOGV只有在宏LOG_NDEBUG定义为0时，即在程序的调试版本中，才是有效的；否则，它只是一个空定义。

这五个宏展开之后，实际上是通过调用日志库liblog提供的函数__android_log_buf_print向Logger日志驱动程序中写入日志记录的。函数__android_log_buf_print的实现可以参考前面4.3小节的内容，这里不再详述。

LOG_EVENT_INT、LOG_EVENT_LONG和LOG_EVENT_STRING

system/core/include/cutils/log.h

```
/*
 * Event log entry types.  These must match up with the declarations in
 * java/android/android/util/EventLog.java.
 */
typedef enum {
    EVENT_TYPE_INT      = 0,
    EVENT_TYPE_LONG     = 1,
    EVENT_TYPE_STRING   = 2,
    EVENT_TYPE_LIST     = 3,
} AndroidEventLogType;


#define LOG_EVENT_INT(_tag, _value) {                                       \
        int intBuf = _value;                                                \
        (void) android_btWriteLog(_tag, EVENT_TYPE_INT, &intBuf,            \
            sizeof(intBuf));                                                \
    }
#define LOG_EVENT_LONG(_tag, _value) {                                      \
        long long longBuf = _value;                                         \
        (void) android_btWriteLog(_tag, EVENT_TYPE_LONG, &longBuf,          \
            sizeof(longBuf));                                               \
    }
#define LOG_EVENT_STRING(_tag, _value)                                      \
    ((void) 0)  /* not implemented -- must combine len with string */
/* TODO: something for LIST */


#define android_btWriteLog(tag, type, payload, len) \
    __android_log_btwrite(tag, type, payload, len)
```

这三个宏是用来写入类型为events的日志记录的。第6行到第9行首先定义了四个枚举值，它们分别用来代表一个整数（int）、长整数（long）、字符串（string）和列表（list）。前面提到，类型

为events的日志记录的内容是由一系列值组成的，这些值是具有类型的，分别对应于EVENT_TYPE_INT、EVENT_TYPE_LONG、EVENT_TYPE_STRING和EVENT_TYPE_LIST四种类型。

宏LOG_EVENT_INT和LOG_EVENT_LONG写入的日志记录的内容分别是一个整数和一个长整数。它们展开之后，实际上是通过调用日志库liblog提供的函数__android_log_btwrite来往Logger日志驱动程序中写入日志记录的。函数__android_log_btwrite的实现可以参考前面4.3小节的内容，这里不再详述。

宏LOG_EVENT_STRING用来往Logger日志驱动程序中写入一条内容为字符串值的日志记录，但是在目前版本的实现中，它只是一个空定义。此外，在目前版本中，系统也没有定义一个用来写入内容为列表的日志记录宏。因此，如果需要往Logger日志驱动程序中写入一条内容为字符串或者列表的日志记录，那么就必须直接使用日志库liblog提供的函数__android_log_bwrite或者__android_log_btwrite。

4.5 Java日志写入接口

Android系统在应用程序框架层中定义了三个Java日志写入接口，它们分别是android.util.Log、android.util.Slog和android.util.EventLog，写入的日志记录类型分别为main、system和events。这三个Java日志写入接口是通过JNI方法来调用日志库liblog提供的函数来实现日志记录的写入功能的。在本节内容中，我们就分别分析这三个Java日志写入接口的实现。

android.util.Log

frameworks/base/core/java/android/util/Log.java

```
01 public final class Log {
02     ......
03
04     /**
05      * Priority constant for the println method; use Log.v.
06          */
07     public static final int VERBOSE = 2;
08
09     /**
10      * Priority constant for the println method; use Log.d.
11          */
12     public static final int DEBUG = 3;
13
14     /**
15      * Priority constant for the println method; use Log.i.
16          */
17     public static final int INFO = 4;
18
19     /**
20      * Priority constant for the println method; use Log.w.
21          */
22     public static final int WARN = 5;
23
24     /**
25      * Priority constant for the println method; use Log.e.
26          */
27     public static final int ERROR = 6;
28
29     /**
30      * Priority constant for the println method.
31          */
```

```
32      public static final int ASSERT = 7;
33
34      ......
35
36      public static int v(String tag, String msg) {
37          return println_native(LOG_ID_MAIN, VERBOSE, tag, msg);
38      }
39
40      public static int d(String tag, String msg) {
41          return println_native(LOG_ID_MAIN, DEBUG, tag, msg);
42      }
43
44      public static int i(String tag, String msg) {
45          return println_native(LOG_ID_MAIN, INFO, tag, msg);
46      }
47
48      public static int w(String tag, String msg) {
49          return println_native(LOG_ID_MAIN, WARN, tag, msg);
50      }
51
52      public static int e(String tag, String msg) {
53          return println_native(LOG_ID_MAIN, ERROR, tag, msg);
54      }
55
56      ......
57
58      /** @hide */ public static native int LOG_ID_MAIN = 0;
59      /** @hide */ public static native int LOG_ID_RADIO = 1;
60      /** @hide */ public static native int LOG_ID_EVENTS = 2;
61      /** @hide */ public static native int LOG_ID_SYSTEM = 3;
62
63      /** @hide */ public static native int println_native(int bufID,
64          int priority, String tag, String msg);
65  }
```

接口android.util.Log提供的日志记录写入成员函数比较多，不过我们只关注常用的成员函数v、d、i、w和e。这些成员函数写入的日志记录的类型都是main，而对应的日志记录的优先级分别为VERBOSE、DEBUG、INFO、WARN和ERROR。它们是通过调用JNI方法println_native来实现日志记录写入功能的，如下所示。

frameworks/base/core/jni/android_util_Log.cpp

```
01 /*
02  * In class android.util.Log:
03  *  public static native int println_native(int buffer, int priority, String tag, String msg)
04  */
05 static jint android_util_Log_println_native(JNIEnv* env, jobject clazz,
06          jint bufID, jint priority, jstring tagObj, jstring msgObj)
07 {
08      const char* tag = NULL;
09      const char* msg = NULL;
10
11      if (msgObj == NULL) {
12          jclass npeClazz;
13
14          npeClazz = env->FindClass("java/lang/NullPointerException");
15          assert(npeClazz != NULL);
16
17          env->ThrowNew(npeClazz, "println needs a message");
18          return -1;
19      }
20
21      if (bufID < 0 || bufID >= LOG_ID_MAX) {
```

```
22          jclass npeClazz;
23
24          npeClazz = env->FindClass("java/lang/NullPointerException");
25          assert(npeClazz != NULL);
26
27          env->ThrowNew(npeClazz, "bad bufID");
28          return -1;
29      }
30
31      if (tagObj != NULL)
32          tag = env->GetStringUTFChars(tagObj, NULL);
33      msg = env->GetStringUTFChars(msgObj, NULL);
34
35      int res = __android_log_buf_write(bufID, (android_LogPriority)priority, tag, msg);
36
37      if (tag != NULL)
38          env->ReleaseStringUTFChars(tagObj, tag);
39      env->ReleaseStringUTFChars(msgObj, msg);
40
41      return res;
42  }
```

在JNI函数android_util_Log_println_native中，第11行到19行代码检查写入的日志记录的内容msgObj是否为null，接着第21行到29行代码检查写入的日志记录的类型值是否位于0和LOG_ID_MAX之间，其中，0、1、2和3四个值表示的日志记录的类型分别为main、radio、events和system。通过这两个合法性检查之后，最后第35行就调用日志库liblog提供的函数__android_log_buf_write来往Logger日志驱动程序中写入日志记录。函数__android_log_buf_write的实现可以参考前面4.3小节的内容，这里不再详述。

android.util.Slog

frameworks/base/core/java/android/util/Slog.java

```
01 /**
02  * @hide
03  */
04 public final class Slog {
05      ......
06
07      public static int v(String tag, String msg) {
08          return Log.println_native(Log.LOG_ID_SYSTEM, Log.VERBOSE, tag, msg);
09      }
10
11      public static int d(String tag, String msg) {
12          return Log.println_native(Log.LOG_ID_SYSTEM, Log.DEBUG, tag, msg);
13      }
14
15      public static int i(String tag, String msg) {
16          return Log.println_native(Log.LOG_ID_SYSTEM, Log.INFO, tag, msg);
17      }
18
19      public static int w(String tag, String msg) {
20          return Log.println_native(Log.LOG_ID_SYSTEM, Log.WARN, tag, msg);
21      }
22
23      public static int e(String tag, String msg) {
24          return Log.println_native(Log.LOG_ID_SYSTEM, Log.ERROR, tag, msg);
25      }
26
27      ......
28  }
```

注意

在接口android.util.Slog定义前面的注释中有一个“@hide”关键字，表示这是一个隐藏接口，只在系统内部使用，应用程序一般不应该使用该接口来写入日志记录。

接口android.util.Slog写入的日志记录的类型为system，它常用的成员函数有v、d、i、w和e，对应的日志记录的优先级分别为VERBOSE、DEBUG、INFO、WARN和ERROR，并且它们都是通过调用接口android.util.Log的JNI方法println_native来实现的。JNI方法println_native的实现已经在前面分析过了，这里不再详述。

android.util.EventLog

frameworks/base/core/java/android/util/EventLog.java

```
01 public class EventLog {
02     ......
03
04     /**
05      * Record an event log message.
06      * @param tag The event type tag code
07      * @param value A value to log
08      * @return The number of bytes written
09      */
10     public static native int writeEvent(int tag, int value);
11
12    /**
13      * Record an event log message.
14      * @param tag The event type tag code
15      * @param value A value to log
16      * @return The number of bytes written
17      */
18     public static native int writeEvent(int tag, long value);
19
20     /**
21      * Record an event log message.
22      * @param tag The event type tag code
23      * @param str A value to log
24      * @return The number of bytes written
25      */
26     public static native int writeEvent(int tag, String str);
27
28     /**
29      * Record an event log message.
30      * @param tag The event type tag code
31      * @param list A list of values to log
32      * @return The number of bytes written
33      */
34     public static native int writeEvent(int tag, Object... list);
35
36     ......
37 }
```

接口android.util.EventLog提供了四个重载版本的JNI方法writeEvent向Logger日志驱动程序中写入类型为events的日志记录，这些日志记录的内容分别为整数、长整数、字符串和列表。

我们首先分析写入整数和长整数类型日志记录的JNI方法writeEvent的实现。

frameworks/base/core/jni/android_util_EventLog.cpp

```
01 /*
02  * In class android.util.EventLog:
03  *  static native int writeEvent(int tag, int value)
04  */
05 static jint android_util_EventLog_writeEvent_Integer(JNIEnv* env, jobject clazz,
06                                                      jint tag, jint value)
```

```
07 {
08     return android_btWriteLog(tag, EVENT_TYPE_INT, &value, sizeof(value));
09 }
10
11 /*
12  * In class android.util.EventLog:
13  *  static native int writeEvent(long tag, long value)
14  */
15 static jint android_util_EventLog_writeEvent_Long(JNIEnv* env, jobject clazz,
16                                                   jint tag, jlong value)
17 {
18     return android_btWriteLog(tag, EVENT_TYPE_LONG, &value, sizeof(value));
19 }
```

它们都是通过调用宏android_btWriteLog向Logger日志驱动程序中写入日志记录的。前者在调用宏android_btWriteLog时，指定第二个参数为EVENT_TYPE_INT，表示要写入的日志记录的内容为一个整数，它的内存布局如图4-8所示。后者在调用宏android_btWriteLog时，指定第二个参数为EVENT_TYPE_LONG，表示要写入的日志记录的内容为一个长整数，它的内存布局如图4-9所示。

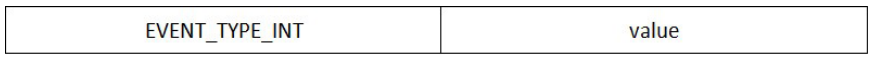


图4-8　内容为整数的日志记录的内存布局

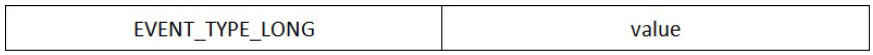


图4-9　内容为长整数的日志记录的内存布局

宏android_btWriteLog的定义如下所示。

system/core/include/cutils/log.h

```
1 #define android_btWriteLog(tag, type, payload, len) \
2     __android_log_btwrite(tag, type, payload, len)
```

它指向了日志库liblog提供的函数__android_log_btwrite。函数__android_log_btwrite的实现可以参考前面4.3小节的内容，这里不再详述。

接下来，我们分析写入字符串类型日志记录的JNI方法writeEvent的实现。

frameworks/base/core/jni/android_util_EventLog.cpp

```
01 /*
02  * In class android.util.EventLog:
03  *  static native int writeEvent(int tag, String value)
04  */
05 static jint android_util_EventLog_writeEvent_String(JNIEnv* env, jobject clazz,
06                                                     jint tag, jstring value) {
07     uint8_t buf[MAX_EVENT_PAYLOAD];
08
09     // Don't throw NPE -- I feel like it's sort of mean for a logging function
10     // to be all crashy if you pass in NULL -- but make the NULL value explicit.
11     const char *str = value != NULL ? env->GetStringUTFChars(value, NULL) : "NULL";
12     jint len = strlen(str);
13     const int max = sizeof(buf) - sizeof(len) - 2;  // Type byte, final newline
14     if (len > max) len = max;
15
16     buf[0] = EVENT_TYPE_STRING;
17     memcpy(&buf[1], &len, sizeof(len));
18     memcpy(&buf[1 + sizeof(len)], str, len);
19     buf[1 + sizeof(len) + len] = '\n';
20
21     if (value != NULL) env->ReleaseStringUTFChars(value, str);
22     return android_bWriteLog(tag, buf, 2 + sizeof(len) + len);
23 }
```

内容为字符串的日志记录的内存布局如图4-10所示。

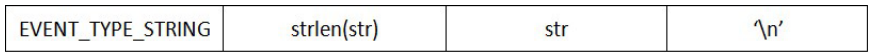


图4-10　内容为字符串的日志记录的内存布局

第一个字段记录日志记录内容的类型为字符串，第二个字段描述该字符串的长度，第三个字段保存的是字符串内容，第四个字段使用特殊字符‘\n’来结束该日志记录。结合这个内存布局图，就不难理解函数android_util_EventLog_writeEvent_String的实现了。

第22行使用宏android_bWriteLog将日志记录写入到Logger日志驱动程序中，它的定义如下所示。

system/core/include/cutils/log.h

```
1 #define android_bWriteLog(tag, payload, len) \
2     __android_log_bwrite(tag, payload, len)
```

它指向了日志库liblog提供的函数__android_log_bwrite。函数__android_log_bwrite的实现可以参考前面4.3小节的内容，这里不再详述。

最后，我们分析写入列表类型日志记录的JNI方法writeEvent的实现。

frameworks/base/core/jni/android_util_EventLog.cpp

```
01 /*
02  * In class android.util.EventLog:
03  *  static native int writeEvent(long tag, Object... value)
04  */
05 static jint android_util_EventLog_writeEvent_Array(JNIEnv* env, jobject clazz,
06                                                     jint tag, jobjectArray value) {
07     if (value == NULL) {
08         return android_util_EventLog_writeEvent_String(env, clazz, tag, NULL);
09     }
10
11     uint8_t buf[MAX_EVENT_PAYLOAD];
12     const size_t max = sizeof(buf) - 1;  // leave room for final newline
13     size_t pos = 2;  // Save room for type tag & array count
14
15     jsize copied = 0, num = env->GetArrayLength(value);
16     for (; copied < num && copied < 255; ++copied) {
17         jobject item = env->GetObjectArrayElement(value, copied);
18         if (item == NULL || env->IsInstanceOf(item, gStringClass)) {
19             if (pos + 1 + sizeof(jint) > max) break;
20             const char *str = item != NULL ? env->GetStringUTFChars((jstring) item, NULL) : "NULL";
21             jint len = strlen(str);
22             if (pos + 1 + sizeof(len) + len > max) len = max - pos - 1 - sizeof(len);
23             buf[pos++] = EVENT_TYPE_STRING;
24             memcpy(&buf[pos], &len, sizeof(len));
25             memcpy(&buf[pos + sizeof(len)], str, len);
26             pos += sizeof(len) + len;
27             if (item != NULL) env->ReleaseStringUTFChars((jstring) item, str);
28         } else if (env->IsInstanceOf(item, gIntegerClass)) {
29             jint intVal = env->GetIntField(item, gIntegerValueID);
30             if (pos + 1 + sizeof(intVal) > max) break;
31             buf[pos++] = EVENT_TYPE_INT;
32             memcpy(&buf[pos], &intVal, sizeof(intVal));
33             pos += sizeof(intVal);
34         } else if (env->IsInstanceOf(item, gLongClass)) {
35             jlong longVal = env->GetLongField(item, gLongValueID);
36             if (pos + 1 + sizeof(longVal) > max) break;
37             buf[pos++] = EVENT_TYPE_LONG;
38             memcpy(&buf[pos], &longVal, sizeof(longVal));
39             pos += sizeof(longVal);
40         } else {
```

```
41              jniThrowException(env,
42                      "java/lang/IllegalArgumentException",
43                      "Invalid payload item type");
44              return -1;
45          }
46          env->DeleteLocalRef(item);
47      }
48
49      buf[0] = EVENT_TYPE_LIST;
50      buf[1] = copied;
51      buf[pos++] = '\n';
52      return android_bWriteLog(tag, buf, pos);
53  }
```

位于列表中的元素的值类型只能为整数、长整数或者字符串；否则，函数就会在第41行到第43行抛出一个异常。一个列表中最多可以包含255个元素，如果超过这个数值，多余的元素就会被忽略。在函数第16行到第47行的for循环中，依次取出列表中的元素，并且根据它们的值类型来组织缓冲区buf的内存布局。如果值类型为整数，那么写入到缓冲区buf的第一个字节设置为一个EVENT_TYPE_INT值，再加上一个整数值；如果值类型为长整数，那么写入到缓冲区buf的内容就为一个EVENT_TYPE_LONG值，再加上一个长整数值；如果值类型为字符串，那么写入到缓冲区buf的内容就为一个EVENT_TYPE_STRING值和一个字符串长度值，再加上字符串的内容。

将列表中的元素都写入到缓冲区buf之后，函数第49行将缓冲区buf的第一个字节设置为EVENT_TYPE_LIST，表示这是一个列表类型的日志记录；接着第50行将缓冲区buf的第二个字节设置为变量copied的值，表示在缓冲区buf中的列表元素个数；最后第51行将缓冲区buf的最后一个字节的内容设置为特殊字符‘\n’，用作该日志记录的结束标志。这时候，缓冲区buf的内存布局就类似于图4-11。

EVENT_TYPE_LIST	copied	EVENT_TYPE_INT\|value	EVENT_TYPE_LONG\|value	EVENT_TYPE_STRING\|strlen(str)\|str	……	'\n'

图4-11　内容为列表的日志记录的内存布局

函数第52行使用了宏android_bWriteLog将日志记录写入到Logger日志驱动程序中。前面提到，宏android_bWriteLog指向的是日志库liblog提供的函数__android_log_bwrite，它的实现可以参考前面4.3小节的内容，这里不再详述。

至此，我们就介绍完了Android系统的日志写入接口。接下来，我们继续分析如何读取并且显示Logger日志驱动程序中的日志记录。

4.6　Logcat工具分析

将日志记录写入到Logger日志驱动程序的目的是要将它们记录起来，等到合适的时候再将它们读取和显示出来，从而帮助我们分析程序的行为。在本节中，我们将通过分析日志查看工具Logcat的实现来学习日志记录的读取过程。

Logcat是内置在Android系统中的一个实用工具，可以在主机上执行adb logcat命令来查看目标设备上的日志记录。例如，首先执行emulator将Android模拟器启动起来，然后执行adb logcat命令激活Android模拟器中的Logcat工具。

```
USER@MACHINE:~/Android$ emulator &
USER@MACHINE:~/Android$ adb logcat
```

最后就可以不断地看到Android模拟器中的日志记录输出。

Logcat工具的用法很丰富，但是本节并不打算详细介绍它的使用方法。如果读者想进一步了解它的用法，可以在Android模拟器启动之后，执行以下命令来获得帮助。

```
USER@MACHINE:~/Android$ adb logcat --help
```

Logcat工具主要涉及的源代码文件的位置如下：

```
~/Android/system/core
----include
     ----cutils
          ----logprint.h
          ----event_tag_map.h
          ----logger.h
     ----android
          ----log.h
----liblog
     ----logprint.c
     ----event_tag_map.c
----logcat
     ----logcat.cpp
```

其中，文件logger.h和log.h定义了一些基础数据结构和宏；文件logprint.h、event_tag_map.h、logprint.c和event_tag_map.c实现在日志库liblog中，它们主要用来处理日志记录的输出；文件logcat.cpp是Logcat工具的源代码实现。

我们首先介绍Logcat工具的基础数据结构，然后分析使用Logcat工具读取和显示Logger日志驱动程序中的日志记录的过程，主要包括三个情景，分别是工具初始化过程、日志记录的读取和输出过程。

4.6.1 基础数据结构

struct logger_entry

system/core/include/cutils/logger.h

```
01 struct logger_entry {
02     uint16_t    len;    /* length of the payload */
03     uint16_t    __pad;  /* no matter what, we get 2 bytes of padding */
04     int32_t     pid;    /* generating process's pid */
05     int32_t     tid;    /* generating process's tid */
06     int32_t     sec;    /* seconds since Epoch */
07     int32_t     nsec;   /* nanoseconds */
08     char        msg[0]; /* the entry's payload */
09 };
10
11 #define LOGGER_ENTRY_MAX_LEN            (4*1024)
12 #define LOGGER_ENTRY_MAX_PAYLOAD        \
13         (LOGGER_ENTRY_MAX_LEN - sizeof(struct logger_entry))
14
```

结构体logger_entry和宏LOGGER_ENTRY_MAX_LEN、LOGGER_ENTRY_MAX_PAYLOAD是从Logger日志驱动程序中提取出来的。在前面的4.2.1小节中，我们已经分析过它们的实现了。其中，结构体logger_entry用来描述一条日志记录；宏LOGGER_ENTRY_MAX_LEN用来描述一条日志记录的最大长度，包括日志记录头和日志记录有效负载两部分内容的长度；宏LOGGER_ENTRY_MAX_PAYLOAD用来描述日志记录有效负载的最大长度。

struct queued_entry_t

system/core/logcat/logcat.cpp

```
01 struct queued_entry_t {
```

```
02      union {
03          unsigned char buf[LOGGER_ENTRY_MAX_LEN + 1] __attribute__((aligned(4)));
04          struct logger_entry entry __attribute__((aligned(4)));
05      };
06      queued_entry_t* next;
07
08      queued_entry_t() {
09          next = NULL;
10      }
11  };
```

结构体queued_entry_t用来描述一个日志记录队列。每一种类型的日志记录都对应有一个日志记录队列。Logcat工具将相同类型的日志记录按照写入时间的先后顺序保存在同一个队列中，这样在输出日志记录时，沿着这个队列就可以按照时间顺序来依次输出系统中的日志信息了。结构体queued_entry_t的第一个成员变量是一个联合体，用来描述一条日志记录的内容。通过这个联合体，Logcat工具就既可以把一条日志记录内容当作一个缓冲区来处理，也可以把它当作一个logger_entry结构体来处理。结构体queued_entry_t的另外一个成员变量next用来连接下一条日志记录，从而形成一个队列。

struct log_device_t

system/core/logcat/logcat.cpp

```
01  struct log_device_t {
02      char* device;
03      bool binary;
04      int fd;
05      bool printed;
06      char label;
07
08      queued_entry_t* queue;
09      log_device_t* next;
10
11      log_device_t(char* d, bool b, char l) {
12          device = d;
13          binary = b;
14          label = l;
15          queue = NULL;
16          next = NULL;
17          printed = false;
18      }
19
20      void enqueue(queued_entry_t* entry) {
21          if (this->queue == NULL) {
22              this->queue = entry;
23          } else {
24              queued_entry_t** e = &this->queue;
25              while (*e && cmp(entry, *e) >= 0) {
26                  e = &((*e)->next);
27              }
28              entry->next = *e;
29              *e = entry;
30          }
31      }
32  };
```

结构体log_device_t用来描述一个日志设备。成员变量device保存的是日志设备文件名称。我们知道，Logger日志驱动程序在初始化时，会创建四个设备文件/dev/log/main、/dev/log/system、/dev/log/radio和/dev/log/events，分别用来代表四个日志设备。成员变量label用来描述日志设备的标号，其中，日志设备/dev/log/main、/dev/log/system、/dev/log/radio和/dev/log/events对应的标号分别为‘m’、‘s’、‘r’和‘e’。成员变量binary是一个布尔值，表示日志记录的内容是否是二进制格式的。我们知

道，只有日志设备/dev/log/events的日志记录内容才是二进制格式的，其余的日志设备的日志记录内容均为文本格式的。成员变量fd是一个文件描述符，它是通过调用函数open来打开相应的日志设备文件得到的，用来从Logger日志驱动程序中读取日志记录。成员变量printed是一个布尔值，用来表示一个日志设备是否已经处于输出状态。成员变量queue用来保存日志设备中的日志记录。成员变量next用来连接下一个日志设备，这样，Logcat工具就可以把所有已经打开的日志设备保存在一个队列中。

结构体log_device_t的成员函数enqueue用来将一条日志记录添加到内部的日志记录队列中。每次往队列中加入一条日志记录时，都会根据它的写入时间来找到它在队列中的位置，然后再将它插入到队列中。两条日志记录的写入时间比较是通过调用函数cmp来实现的，它的定义如下所示。

system/core/logcat/logcat.cpp

```
1 static int cmp(queued_entry_t* a, queued_entry_t* b) {
2     int n = a->entry.sec - b->entry.sec;
3     if (n != 0) {
4         return n;
5     }
6     return a->entry.nsec - b->entry.nsec;
7 }
```

它先比较日志写入时间的秒数。如果相等，再比较日志写入时间的毫秒数；否则，就可以得到比较结果。如果函数cmp的返回值大于0，就表示日志记录a的写入时间比日志记录b晚；如果函数cmp的返回值等于0，就表示日志记录a和b的写入时间相等；如果函数cmp的返回值小于0，就表示日志记录a的写入时间比日志记录b早。

enum android_LogPriority

system/core/include/android/log.h

```
01 /*
02  * Android log priority values, in ascending priority order.
03  */
04 typedef enum android_LogPriority {
05     ANDROID_LOG_UNKNOWN = 0,
06     ANDROID_LOG_DEFAULT,    /* only for SetMinPriority() */
07     ANDROID_LOG_VERBOSE,
08     ANDROID_LOG_DEBUG,
09     ANDROID_LOG_INFO,
10     ANDROID_LOG_WARN,
11     ANDROID_LOG_ERROR,
12     ANDROID_LOG_FATAL,
13     ANDROID_LOG_SILENT,     /* only for SetMinPriority(); must be last */
14 } android_LogPriority;
```

枚举android_LogPriority用来描述日志记录的优先级，它一共有九个值，分别表示九个优先级。

enum AndroidLogPrintFormat

system/core/include/cutils/logprint.h

```
01 typedef enum {
02     FORMAT_OFF = 0,
03     FORMAT_BRIEF,
04     FORMAT_PROCESS,
05     FORMAT_TAG,
06     FORMAT_THREAD,
07     FORMAT_RAW,
08     FORMAT_TIME,
09     FORMAT_THREADTIME,
10     FORMAT_LONG,
11 } AndroidLogPrintFormat;
```

枚举AndroidLogPrintFormat用来描述日志记录的输出格式，它一共有九个值，分别表示九种不同的输出格式。在分析日志记录的输出过程时，我们再分析它们的含义。

struct AndroidLogEntry_t

system/core/include/cutils/logprint.h

```
01 typedef struct AndroidLogEntry_t {
02     time_t tv_sec;
03     long tv_nsec;
04     android_LogPriority priority;
05     pid_t pid;
06     pthread_t tid;
07     const char * tag;
08     size_t messageLen;
09     const char * message;
10 } AndroidLogEntry;
```

与结构体logger_entry一样，结构体AndroidLogEntry_t也是用来描述一条日志记录的，不过，它描述的日志记录经过了解析，即已经将它的优先级、标签和内容都解析出来，分别保存在成员变量priority、tag和message中。

struct FilterInfo_t

system/core/liblog/logprint.c

```
1 typedef struct FilterInfo_t {
2     char *mTag;
3     android_LogPriority mPri;
4     struct FilterInfo_t *p_next;
5 } FilterInfo;
```

结构体FilterInfo_t用来描述一个日志记录输出过滤器。成员变量mTag和mPri分别表示要过滤的日志记录的标签和优先级。当一条日志记录的标签等于mTag时，如果它的优先级大于等于mPri，那么它就会被输出，否则就会被忽略。成员变量p_next用来连接下一个日志记录输出过滤器，这样就可以将所有的日志记录输出过滤器连接在一起形成一个队列。

struct AndroidLogFormat_t

system/core/liblog/logprint.c

```
1 struct AndroidLogFormat_t {
2     android_LogPriority global_pri;
3     FilterInfo *filters;
4     AndroidLogPrintFormat format;
5 };
```

结构体AndroidLogFormat_t用来保存日志记录的输出格式以及输出过滤器。其中，成员变量global_pri是一个全局设置的默认日志记录输出过滤优先级，成员变量filters是一个日志记录输出过滤器列表，而成员变量format用来保存具体的日志记录输出格式。当日志记录输出过滤器列表filters中的某一个过滤器的过滤优先级被设置为默认优先级ANDROID_LOG_DEFAULT时，系统就会将它的过滤优先级修改为global_pri。

struct EventTag

system/core/liblog/event_tag_map.c

```
1 typedef struct EventTag {
2     unsigned int    tagIndex;
3     const char*     tagStr;
4 } EventTag;
```

结构体EventTag用来描述类型为events的日志记录的标签号，每一个标签号（tagIndex）都对应有

一个文本描述字符串（tagStr）。这些对应关系是通过解析目标设备上的/system/etc/event-log-tags文件得到的。

struct EventTagMap

system/core/liblog/event_tag_map.c

```
1 struct EventTagMap {
2     /* memory-mapped source file; we get strings from here */
3     void*           mapAddr;
4     size_t          mapLen;
5
6     /* array of event tags, sorted numerically by tag index */
7     EventTag*       tagArray;
8     int             numTags;
9 };
```

结构体EventTagMap用来描述类型为events的日志记录的内容格式，它同样是通过解析目标设备上的/system/etc/event-log-tags文件得到的。Logcat工具在打开目标设备上的/system/etc/event-log-tags文件时，会把它的内容映射到内存中，其中，成员变量mapAddr就指向这块内存的起始地址；成员变量mapLen表示该内存的大小；成员变量tagArray是一个EventTag类型的数组，数组的大小由成员变量numTags来描述。

4.6.2 初始化过程

Logcat工具的初始化过程是从文件logcat.cpp中的函数main开始的，它会打开日志设备和解析命令行参数。这个函数的实现代码比较长，我们分段来阅读。

system/core/logcat/logcat.cpp

```
001 static AndroidLogFormat * g_logformat;
002 static bool g_nonblock = false;
003 static int g_tail_lines = 0;
004
005 static const char * g_outputFileName = NULL;
006 static int g_logRotateSizeKBytes = 0;                    // 0 means "no log rotation"
007 static int g_maxRotatedLogs = DEFAULT_MAX_ROTATED_LOGS; // 0 means "unbounded"
008 static int g_outFD = -1;
009 static off_t g_outByteCount = 0;
010 static int g_printBinary = 0;
011 static int g_devCount = 0;
012
013 static EventTagMap* g_eventTagMap = NULL;
014
015 int main(int argc, char **argv)
016 {
017     int err;
018     int hasSetLogFormat = 0;
019     ......
020     const char *forceFilters = NULL;
021     log_device_t* devices = NULL;
022     log_device_t* dev;
023     bool needBinary = false;
024
025     g_logformat = android_log_format_new();
```

第25行调用函数android_log_format_new来创建一个全局的日志记录输出格式和输出过滤器对象g_logformat。

system/core/liblog/logprint.c

```
01 AndroidLogFormat *android_log_format_new()
02 {
03     AndroidLogFormat *p_ret;
04
05     p_ret = calloc(1, sizeof(AndroidLogFormat));
06
07     p_ret->global_pri = ANDROID_LOG_VERBOSE;
08     p_ret->format = FORMAT_BRIEF;
09
10     return p_ret;
11 }
```

从函数android_log_format_new的实现就可以看出，全局变量g_logformat指定的日志记录输出格式为FORMAT_BRIEF，而指定的全局默认日志记录过滤优先级为ANDROID_LOG_VERBOSE。

回到函数main中，我们继续往下分析。

system/core/logcat/logcat.cpp

```
026     for (;;) {
027         int ret;
028
029         ret = getopt(argc, argv, "cdt:gsQf:r::n:v:b:B");
030         ......
031
032         switch(ret) {
033         ......
034         case 'd':
035                 g_nonblock = true;
036         break;
037         case 't':
038                 g_nonblock = true;
039                 g_tail_lines = atoi(optarg);
040         break;
041         ......
042         case 'b': {
043                 char* buf = (char*) malloc(strlen(LOG_FILE_DIR) + strlen(optarg) + 1);
044                 strcpy(buf, LOG_FILE_DIR);
045                 strcat(buf, optarg);
046
047                 bool binary = strcmp(optarg, "events") == 0;
048                 if (binary) {
049                     needBinary = true;
050                 }
051
052                 if (devices) {
053                     dev = devices;
054                     while (dev->next) {
055                         dev = dev->next;
056                     }
057                     dev->next = new log_device_t(buf, binary, optarg[0]);
058                 } else {
059                     devices = new log_device_t(buf, binary, optarg[0]);
060                 }
061                 android::g_devCount++;
062             }
063         break;
064         case 'B':
065                 android::g_printBinary = 1;
066         break;
067         case 'f':
068                 // redirect output to a file
069
```

```
070                 android::g_outputFileName = optarg;
071
072         break;
073         case 'r':
074                 if (optarg == NULL) {
075                     android::g_logRotateSizeKBytes
076                                 = DEFAULT_LOG_ROTATE_SIZE_KBYTES;
077                 } else {
078                     long logRotateSize;
079                     char *lastDigit;
080
081                     if (!isdigit(optarg[0])) {
082                         ......
083                     }
084                     android::g_logRotateSizeKBytes = atoi(optarg);
085                 }
086         break;
087         case 'n':
088                 if (!isdigit(optarg[0])) {
089                     ......
090                 }
091
092                 android::g_maxRotatedLogs = atoi(optarg);
093         break;
094         case 'v':
095                 err = setLogFormat(optarg);
096                 ......
097
098                 hasSetLogFormat = 1;
099         break;
100         ......
101         }
102     }
```

函数第26行到第102行的for循环依次对命令行参数进行解析。命令行参数的字符串解析是通过调用函数getopt来实现的，它的返回值ret表示命令行中的一个选项，而选项对应的值保存在变量optarg中。接下来，我们就分别对命令行中的选项d、t、b、B、f、r、n和v进行介绍。

如果使用选项d来启动Logcat工具，那么函数第35行就会把全局变量g_nonblock的值设置为true，表示当Logger日志驱动程序中没有日志记录可读时，Logcat工具就直接退出。

如果使用选项t来启动Logcat工具，那么函数第39行就会将选项后面的数字保存在全局变量g_tail_lines中，同时第38行会将全局变量g_nonblock的值设置为true。其中，全局变量g_tail_lines表示Logcat工具每次在输出日志记录时，只输出最新的日志记录条数。

如果使用选项b来启动Logcat工具，那么函数第52行到第60行代码就会将选项后面的字符串取出来，并且为它创建一个log_device_t结构体，表示Logcat工具要打开的日志设备。选项后面的字符串的取值为“main”、“radio”或者“events”，表示要打开的日志设备分别为/dev/log/main、/dev/log/radio或者/dev/log/events。当选项后面的字符串为“events”时，函数第49行会将变量needBinary的值设置为true，表示Logcat工具要解析目标设备上的/system/etc/event-log-tags文件的内容，以便可以将日志设备 /dev/log/events的内容从二进制格式转换为文本格式。每次创建一个log_device_t结构体时，全局变量g_devCount的值就会加1，表示Logcat工具所打开的日志设备个数。

注意

如果在启动Logcat工具时，没有使用选项b，那么Logcat工具默认打开的日志设备就为/dev/log/main。

如果使用选项B来启动Logcat工具，那么函数第65行就会将全局变量g_printBinary的值设置为1，

表示要以二进制格式来输出日志记录。这时候就不需要对日志记录的内容进行解析了。

如果使用选项f来启动Logcat工具，那么函数第70行就会将选项后面的字符串取出来，并且保存在全局变量g_outputFileName中，用作日志记录的输出文件名称。当输入到文件g_outputFileName的日志记录的大小（单位是字节）达到设定的量时，Logcat工具就会将接下来的日志记录输出到另外一个文件中，直到这个文件的日志记录的大小也达到设定的量为止。用作日志记录输出的文件的命名方式是遵循一定的规律的。例如，假设选项f后面的字符串为 “logfile”，那么第一个日志记录输出文件的名称就为“logfile”，接下来的日志记录输出文件的名称依次为“logfile.1”、“logfile.2”、“logfile.3”等。日志记录输出文件的大小由选项r来指定，而日志记录输出文件的个数由选项n来指定。当日志记录输出文件的个数达到设定的值时，并且每一个日志记录输出文件的大小也达到设定的值时，Logcat工具就会循环使用已有的日志记录输出文件，即将原来的日志记录输出文件的内容擦掉，然后将新的日志记录写入到这些文件中。

如果使用选项r来启动Logcat工具，那么函数第84行就会将选项后面的数字保存在全局变量g_logRotateSizeKBytes中，用来表示每一个日志记录输出文件的最大容量。如果该选项后面没有指定数字，那么全局变量g_logRotateSizeKBytes的值就默认设置为DEFAULT_LOG_ROTATE_SIZE_KBYTES。

system/core/logcat/logcat.cpp

```
1 #define DEFAULT_LOG_ROTATE_SIZE_KBYTES 16
```

全局变量g_logRotateSizeKBytes的值被初始化为0，表示所有的日志记录输出文件的容量没有限制，这会导致所有的日志记录都输出到文件g_outputFileName中。

如果使用选项n来启动Logcat工具，那么函数第92行就会将选项后面的数字保存在全局变量g_maxRotatedLogs中，表示日志记录输出文件的最大个数。全局变量g_maxRotateLogs的值被初始化为DEFAULT_MAX_ROTATED_LOGS。

system/core/logcat/logcat.cpp

```
1 #define DEFAULT_MAX_ROTATED_LOGS 16
```

如果使用选项v来启动Logcat工具，那么函数第95行就会调用函数setLogFormat将选项后面的字符串转换为相应的AndroidLogPrintFormat枚举值。该选项后面的字符串取值为“brief”、“process”、“tag”、“thread”、“raw”、“time”、“threadtime”或者“long”，分别对应于枚举AndroidLogPrintFormat中的每一个值。

函数setLogFormat的实现如下所示。

system/core/logcat/logcat.cpp

```
01 static int setLogFormat(const char * formatString)
02 {
03     static AndroidLogPrintFormat format;
04
05     format = android_log_formatFromString(formatString);
06
07     if (format == FORMAT_OFF) {
08         // FORMAT_OFF means invalid string
09         return -1;
10     }
11
12     android_log_setPrintFormat(g_logformat, format);
13
14     return 0;
15 }
```

第5行调用函数android_log_formatFromString将字符串formatString转换为一个AndroidLogPrintFormat枚举值。

system/core/liblog/logprint.c

```
01 AndroidLogPrintFormat android_log_formatFromString(const char * formatString)
02 {
03     static AndroidLogPrintFormat format;
04
05     if (strcmp(formatString, "brief") == 0) format = FORMAT_BRIEF;
06     else if (strcmp(formatString, "process") == 0) format = FORMAT_PROCESS;
07     else if (strcmp(formatString, "tag") == 0) format = FORMAT_TAG;
08     else if (strcmp(formatString, "thread") == 0) format = FORMAT_THREAD;
09     else if (strcmp(formatString, "raw") == 0) format = FORMAT_RAW;
10     else if (strcmp(formatString, "time") == 0) format = FORMAT_TIME;
11     else if (strcmp(formatString, "threadtime") == 0) format = FORMAT_THREADTIME;
12     else if (strcmp(formatString, "long") == 0) format = FORMAT_LONG;
13     else format = FORMAT_OFF;
14
15     return format;
16 }
```

如果参数formatString是一个非法字符串，那么函数android_log_formatFromString就会将它转换为一个FORMAT_OFF值；否则，就通过第5行到第12行代码来得到正确的AndroidLogPrintFormat枚举值。

回到函数setLogFormat中，将前面得到的日志记录输出格式保存在变量format中，接着第12行继续调用函数android_log_setPrintFormat将变量format的值设置到全局变量g_logformat的成员变量format中，用来描述Logcat工具的日志记录输出格式。

system/core/liblog/logprint.c

```
1 void android_log_setPrintFormat(AndroidLogFormat *p_format,
2         AndroidLogPrintFormat format)
3 {
4     p_format->format=format;
5 }
```

回到main函数中，如果指定了选项v，并且成功地对它进行了解析，那么函数第98行就会将变量hasSetLogFormat的值设置为1，表示设置了Logcat工具的日志记录输出格式。

函数main解析完成命令行参数后，继续往下执行。

system/core/logcat/logcat.cpp

```
103     if (!devices) {
104         devices = new log_device_t(strdup("/dev/"LOGGER_LOG_MAIN), false, 'm');
105         android::g_devCount = 1;
106         int accessmode =
107                     (mode & O_RDONLY) ? R_OK : 0
108                   | (mode & O_WRONLY) ? W_OK : 0;
109         // only add this if it's available
110         if (0 == access("/dev/"LOGGER_LOG_SYSTEM, accessmode)) {
111             devices->next = new log_device_t(strdup("/dev/"LOGGER_LOG_SYSTEM), false, 's');
112             android::g_devCount++;
113         }
114     }
```

如果启动Logcat工具时，没有指定选项b，那么变量devices的值就会等于NULL。这时候Logcat工具就会默认打开日志设备/dev/log/main来读取它的日志记录。如果日志设备/dev/log/system也存在，那么Logcat工具也会一起打开它来读取日志记录。

打开要读取日志记录的日志设备之后，函数main继续往下执行。

system/core/logcat/logcat.cpp

```
115     android::setupOutput();
```

第115行调用函数setupOutput来设置日志记录是输出到文件中还是打印到标准输出中，它的实现如下所示。

system/core/logcat/logcat.cpp

```
01 static void setupOutput()
02 {
03
04     if (g_outputFileName == NULL) {
05         g_outFD = STDOUT_FILENO;
06
07     } else {
08         struct stat statbuf;
09
10         g_outFD = openLogFile(g_outputFileName);
11
12         if (g_outFD < 0) {
13             perror ("couldn't open output file");
14             exit(-1);
15         }
16
17         fstat(g_outFD, &statbuf);
18
19         g_outByteCount = statbuf.st_size;
20     }
21 }
```

第4行判断全局变量g_outputFileName的值是否等于NULL。如果是，则说明要把日志记录打印到标准输出中，因此就将全局变量g_outFD指向标准输出文件描述符STDOUT_FILENO；否则，就要将日志记录输出到文件g_outputFileName中。如果是将日志记录输出到文件中，那么函数第10行调用openLogFile函数以APPEND方式打开日志记录输出文件g_outputFileName。

system/core/logcat/logcat.cpp

```
1 static int openLogFile (const char *pathname)
2 {
3     return open(g_outputFileName, O_WRONLY | O_APPEND | O_CREAT, S_IRUSR | S_IWUSR);
4 }
```

回到函数setupOutput中，第17行获得当前日志记录输出文件g_outputFileName的大小，并且将它保存在全局变量g_outByteCount中。接下来Logcat工具往文件g_outputFileName输出日志记录时，就会相应地增加全局变量g_outByteCount的值。当全局变量g_outByteCount的值达到g_logRotateSizeKBytes时，Logcat工具就会将后面的日志记录输出到另外一个文件中。

回到函数main中，继续往下执行。

system/core/logcat/logcat.cpp

```
116     if (hasSetLogFormat == 0) {
117         const char* logFormat = getenv("ANDROID_PRINTF_LOG");
118
119         if (logFormat != NULL) {
120             err = setLogFormat(logFormat);
121             ......
122         }
123     }
```

如果启动Logcat工具时，没有指定选项v，那么变量hasSetLogFormat的值就会等于0，表示没有指定日志记录的输出格式。这时候Logcat工具就会检查环境变量ANDROID_PRINTF_LOG的值。如果环境变量ANDROID_PRINTF_LOG存在，那么就会调用函数setLogFormat将它的值作为Logcat工具的日志记录输出格式。

设置好日志记录的输出格式之后，函数main继续往下执行。

system/core/logcat/logcat.cpp

```
124     if (forceFilters) {
125         err = android_log_addFilterString(g_logformat, forceFilters);
126         ......
127     } else if (argc == optind) {
128         // Add from environment variable
129         char *env_tags_orig = getenv("ANDROID_LOG_TAGS");
130
131         if (env_tags_orig != NULL) {
132             err = android_log_addFilterString(g_logformat, env_tags_orig);
133             ......
134         }
135     } else {
136         // Add from commandline
137         for (int i = optind ; i < argc ; i++) {
138             err = android_log_addFilterString(g_logformat, argv[i]);
139             ......
140         }
141     }
```

在启动Logcat工具时，我们可以指定一个不公开的选项Q，这时候Logcat工具就会去读取目标设备上的/proc/cmdline文件，检查里面有没有设置日志记录输出过滤器。如果设置了，那么变量forceFilters就指向这个日志记录输出过滤器，因此，函数第125行就会调用函数android_log_addFilterString将它增加到Logcat工具的日志记录输出过滤器列表中。

如果启动Logcat工具时，没有指定选项Q，那么Logcat工具就会继续检查命令行是否还带有其他额外的参数。如果没有，即第127行的if语句为true，那么Logcat工具就会将环境变量ANDROID_LOG_TAGS的值作为日志记录输出的一个过滤器来使用。

如果启动Logcat工具时，没有指定选项Q，但是命令行带有额外的参数，那么函数第137行到第140行的for循环就会检查这些额外的参数是否是一个日志记录输出过滤表达式。例如，当使用下面的命令来启动Logcat工具时：

```
USER@MACHINE:~/Android$ adb logcat LOGTAG:I
```

最后一个参数“LOGTAG:I”就称为日志记录输出过滤表达式。日志记录输出过滤表达式的格式为“<tag>[:priority]”，其中，tag为任意字符串，表示一个日志记录标签；priority是一个字符，表示一个日志记录优先级。合法的priority字符为数字‘0’到‘9’，或者字母‘v’、‘d’、‘i’、‘w’、‘f’、‘s’和‘*’，它们的具体含义在后面会进一步解释。这时候函数第138行就会调用函数android_log_addFilterString来解析这些日志记录输出过滤表达式，并且将它增加到Logcat工具的日志记录输出过滤器列表中。

函数android_log_addFilterString的实现如下所示。

system/core/liblog/logprint.c

```
01 /**
02  * filterString: a comma/whitespace-separated set of filter expressions
03  *
04  * eg "AT:d *:i"
05  *
06  * returns 0 on success and -1 on invalid expression
07  *
08  * Assumes single threaded execution
09  *
10  */
11
12 int android_log_addFilterString(AndroidLogFormat *p_format,
13         const char *filterString)
14 {
```

```
    char *filterStringCopy = strdup (filterString);
    char *p_cur = filterStringCopy;
    char *p_ret;
    int err;

    // Yes, I'm using strsep
    while (NULL != (p_ret = strsep(&p_cur, " \t,"))) {
        // ignore whitespace-only entries
        if(p_ret[0] != '\0') {
            err = android_log_addFilterRule(p_format, p_ret);

            if (err < 0) {
                goto error;
            }
        }
    }

    free (filterStringCopy);
    return 0;
error:
    free (filterStringCopy);
    return -1;
}
```

第二个参数filterString可以同时包括若干个日志记录输出过滤表达式，它们以空格、制表符或者逗号分割。每一个日志记录输出过滤表达式都是使用函数android_log_addFilterRule来解析的，它的实现如下所示。

system/core/liblog/logprint.c

```
/**
 * filterExpression: a single filter expression
 * eg "AT:d"
 *
 * returns 0 on success and -1 on invalid expression
 *
 * Assumes single threaded execution
 */

int android_log_addFilterRule(AndroidLogFormat *p_format,
        const char *filterExpression)
{
    size_t i=0;
    size_t tagNameLength;
    android_LogPriority pri = ANDROID_LOG_DEFAULT;

    tagNameLength = strcspn(filterExpression, ":");

    if (tagNameLength == 0) {
        goto error;
    }

    if(filterExpression[tagNameLength] == ':') {
        pri = filterCharToPri(filterExpression[tagNameLength+1]);

        if (pri == ANDROID_LOG_UNKNOWN) {
            goto error;
        }
    }

    if(0 == strncmp("*", filterExpression, tagNameLength)) {
        // This filter expression refers to the global filter
        // The default level for this is DEBUG if the priority
```

```
34          // is unspecified
35          if (pri == ANDROID_LOG_DEFAULT) {
36              pri = ANDROID_LOG_DEBUG;
37          }
38
39          p_format->global_pri = pri;
40      } else {
41          // for filter expressions that don't refer to the global
42          // filter, the default is verbose if the priority is unspecified
43          if (pri == ANDROID_LOG_DEFAULT) {
44              pri = ANDROID_LOG_VERBOSE;
45          }
46
47          char *tagName;
48
49 // Presently HAVE_STRNDUP is never defined, so the second case is always taken
50 // Darwin doesn't have strnup, everything else does
51 #ifdef HAVE_STRNDUP
52          tagName = strndup(filterExpression, tagNameLength);
53 #else
54          //a few extra bytes copied...
55          tagName = strdup(filterExpression);
56          tagName[tagNameLength] = '\0';
57 #endif /*HAVE_STRNDUP*/
58
59          FilterInfo *p_fi = filterinfo_new(tagName, pri);
60          free(tagName);
61
62          p_fi->p_next = p_format->filters;
63          p_format->filters = p_fi;
64      }
65
66      return 0;
67 error:
68      return -1;
69 }
```

第17行取得日志记录输出过滤表达式中冒号的位置，接着第24行就调用函数filterCharToPri将冒号后面的字符转换为android_LogPriority枚举值。

system/core/liblog/logprint.c

```
01 /*
02  * Note: also accepts 0-9 priorities
03  * returns ANDROID_LOG_UNKNOWN if the character is unrecognized
04  */
05 static android_LogPriority filterCharToPri (char c)
06 {
07      android_LogPriority pri;
08
09      c = tolower(c);
10
11      if (c >= '0' && c <= '9') {
12          if (c >= ('0'+ANDROID_LOG_SILENT)) {
13              pri = ANDROID_LOG_VERBOSE;
14          } else {
15              pri = (android_LogPriority)(c - '0');
16          }
17      } else if (c == 'v') {
18          pri = ANDROID_LOG_VERBOSE;
19      } else if (c == 'd') {
20          pri = ANDROID_LOG_DEBUG;
21      } else if (c == 'i') {
22          pri = ANDROID_LOG_INFO;
```

```
23      } else if (c == 'w') {
24          pri = ANDROID_LOG_WARN;
25      } else if (c == 'e') {
26          pri = ANDROID_LOG_ERROR;
27      } else if (c == 'f') {
28          pri = ANDROID_LOG_FATAL;
29      } else if (c == 's') {
30          pri = ANDROID_LOG_SILENT;
31      } else if (c == '*') {
32          pri = ANDROID_LOG_DEFAULT;
33      } else {
34          pri = ANDROID_LOG_UNKNOWN;
35      }
36
37      return pri;
38  }
```

从函数filterCharToPri的实现就可以看出，字符“v”、“d”、“i”、“w”、“e”、“f”、“s”和“*”对应的日志记录优先级分别为ANDROID_LOG_VERBOSE、ANDROID_LOG_DEBUG、ANDROID_LOG_INFO、ANDROID_LOG_WARN、ANDROID_LOG_ERROR、ANDROID_LOG_FATAL、ANDROID_LOG_SILENT和ANDROID_LOG_DEFAULT。此外，字符“0”到“7”对应的日志记录优先级分别为ANDROID_LOG_UNKNOWN、ANDROID_LOG_DEFAULT、ANDROID_LOG_VERBOSE、ANDROID_LOG_DEBUG、ANDROID_LOG_INFO、ANDROID_LOG_WARN、ANDROID_LOG_ERROR和ANDROID_LOG_FATAL，而字符“8”和“9”对应的日志记录优先级均为ANDROID_LOG_VERBOSE。

回到函数android_log_addFilterRule中，如果日志记录输出过滤表达式中的日志记录标签值为“*”，即第31行的if语句为true，就表示要将它的日志记录优先级设置为全局的日志记录过滤优先级；否则，就为该日志记录输出过滤表达式创建一个日志记录输出过滤器，并且增加到Logcat工具的日志记录输出过滤器列表中，如第59行到第63行代码所示。

设置好日志记录输出过滤器列表之后，回到函数main中，继续往下执行。

system/core/logcat/logcat.cpp

```
142     dev = devices;
143     while (dev) {
144         dev->fd = open(dev->device, mode);
145         ......
146
147         dev = dev->next;
148     }
```

函数第143行到148行的while循环依次调用函数open来打开保存在devices列表中的各个日志设备，并且将得到的文件描述符保存在对应的日志设备的成员变量fd中。

日志设备打开之后，函数main继续往下执行。

system/core/logcat/logcat.cpp

```
149     if (needBinary)
150         android::g_eventTagMap = android_openEventTagMap(EVENT_TAG_MAP_FILE);
```

函数第149行判断打开的日志设备的日志记录格式是否包含二进制格式，即是否打开了日志设备/dev/log/events。如果是，即第149行的if语句为true，那么第150行就会调用函数android_openEventTagMap来解析目标设备上的/system/etc/event-log-tags文件，因为Logcat工具需要根据它的内容来解析类型为events的日志记录。函数android_openEventTagMap的返回值是一个EventTagMap结构体，保存在全局变量g_eventTagMap中。

EVENT_TAG_MAP_FILE是一个宏定义，它的值为“/system/etc/event-log-tags”，如下所示。

system/core/include/cutils/event_tag_map.h

```
1 #define EVENT_TAG_MAP_FILE  "/system/etc/event-log-tags"
```

函数android_openEventTagMap的实现如下所示。

system/core/liblog/event_tag_map.c

```
01 /*
02  * Open the map file and allocate a structure to manage it.
03  *
04  * We create a private mapping because we want to terminate the log tag
05  * strings with '\0'.
06  */
07 EventTagMap* android_openEventTagMap(const char* fileName)
08 {
09     EventTagMap* newTagMap;
10     off_t end;
11     int fd = -1;
12
13     newTagMap = calloc(1, sizeof(EventTagMap));
14     if (newTagMap == NULL)
15         return NULL;
16
17     fd = open(fileName, O_RDONLY);
18     if (fd < 0) {
19         fprintf(stderr, "%s: unable to open map '%s': %s\n",
20             OUT_TAG, fileName, strerror(errno));
21         goto fail;
22     }
23
24     end = lseek(fd, 0L, SEEK_END);
25     (void) lseek(fd, 0L, SEEK_SET);
26     if (end < 0) {
27         fprintf(stderr, "%s: unable to seek map '%s'\n", OUT_TAG, fileName);
28         goto fail;
29     }
30
31     newTagMap->mapAddr = mmap(NULL, end, PROT_READ | PROT_WRITE, MAP_PRIVATE,
32                               fd, 0);
33     if (newTagMap->mapAddr == MAP_FAILED) {
34         fprintf(stderr, "%s: mmap(%s) failed: %s\n",
35             OUT_TAG, fileName, strerror(errno));
36         goto fail;
37     }
38
39     newTagMap->mapLen = end;
40
41     if (processFile(newTagMap) != 0)
42         goto fail;
43
44     return newTagMap;
45
46 fail:
47     android_closeEventTagMap(newTagMap);
48     if (fd >= 0)
49         close(fd);
50     return NULL;
51 }
```

第13行首先创建一个EventTagMap结构体对象，接着第17行打开文件/system/etc/event-log-tags，第24行得到该文件的大小，第31行将该文件的内容映射到内存中，最后第41行调用函数processFile来解析文件/system/etc/event-log-tags中的内容。

system/core/liblog/event_tag_map.c

```
01 /*
02  * Crunch through the file, parsing the contents and creating a tag index.
03  */
04 static int processFile(EventTagMap* map)
05 {
06     EventTag* tagArray = NULL;
07
08     /* get a tag count */
09     map->numTags = countMapLines(map);
10     if (map->numTags < 0)
11         return -1;
12
13     //printf("+++ found %d tags\n", map->numTags);
14
15     /* allocate storage for the tag index array */
16     map->tagArray = calloc(1, sizeof(EventTag) * map->numTags);
17     if (map->tagArray == NULL)
18         return -1;
19
20     /* parse the file, null-terminating tag strings */
21     if (parseMapLines(map) != 0) {
22         fprintf(stderr, "%s: file parse failed\n", OUT_TAG);
23         return -1;
24     }
25
26     /* sort the tags and check for duplicates */
27     if (sortTags(map) != 0)
28         return -1;
29
30     return 0;
31 }
```

第9行调用函数countMapLines来计算文件/system/etc/event-log-tags的行数，接着第16行根据这个行数来分配一个EventTag结构体数组，其中，每一行的内容都对应一个EventTag结构体。第21行调用函数parseMapLines来解析文件/system/etc/event-log-tags的内容。函数countMapLines主要是根据文件中的换行符‘\n’来计算行数，而且会忽略文件中的空白行和注释行。函数parseMapLines对文件中的每一行内容进行解析，从而得到类型为events的日志记录标签描述表。下面我们就通过一个例子来说明函数parseMapLines的工作原理。

当目标设备上的文件/system/etc/event-log-tags包含以下一行内容时：

```
2722 battery_level (level|1|6),(voltage|1|1),(temperature|1|1)
```

函数parseMapLines就会将字符串“2722”转换为数字2722，并且将它保存在一个EventTag结构体的成员变量tagIndex中，然后再将字符“battery_level”在内存中的地址保存在同一个EventTag结构体的成员变量tagStr中，从而将日志记录标签号2722与描述字符串“battery_level”关联起来。

函数parseMapLines执行完成之后，我们就得到了类型为events的日志记录的日志标签描述表，它保存在一个EventTagMap结构体内部的数组tagArray中。为了使后面能够根据日志标签号快速找到对应的日志标签描述字符串，函数processFile的第27行调用函数sortTags对这个数组进行排序。

system/core/liblog/event_tag_map.c

```
01 /*
02  * Sort the EventTag array so we can do fast lookups by tag index.  After
03  * the sort we do a quick check for duplicate tag indices.
04  *
05  * Returns 0 on success.
06  */
07 static int sortTags(EventTagMap* map)
```

```
08 {
09     int i;
10
11     qsort(map->tagArray, map->numTags, sizeof(EventTag), compareEventTags);
12
13     for (i = 1; i < map->numTags; i++) {
14         if (map->tagArray[i].tagIndex == map->tagArray[i-1].tagIndex) {
15             fprintf(stderr, "%s: duplicate tag entries (%d:%s and %d:%s)\n",
16                 OUT_TAG,
17                 map->tagArray[i].tagIndex, map->tagArray[i].tagStr,
18                 map->tagArray[i-1].tagIndex, map->tagArray[i-1].tagStr);
19             return -1;
20         }
21     }
22
23     return 0;
24 }
```

第11行调用快排函数qsort对该日志标签描述数组进行排序，指定的比较函数为compareEventTags，它的实现如下所示。

system/core/liblog/event_tag_map.c

```
01 /*
02  * Compare two EventTags.
03  */
04 static int compareEventTags(const void* v1, const void* v2)
05 {
06     const EventTag* tag1 = (const EventTag*) v1;
07     const EventTag* tag2 = (const EventTag*) v2;
08
09     return tag1->tagIndex - tag2->tagIndex;
10 }
```

从这里就可以看出，日志标签描述数组的元素是按照日志标签号从小到大排列的。

回到函数sortTags中，第13行到第21行的for循环检查前面获得的日志标签描述数组的合法性，即检查它里面是否有重复的日志标签号。如果有，就会返回错误码-1，表示目标设备上的文件/system/etc/event-log-tags的内容有问题。

解析完成目标设备上的文件/system/etc/event-log-tags之后，回到函数main中，继续往下执行。

system/core/logcat/logcat.cpp

```
151     android::readLogLines(devices);
152
153     return 0;
154 }
```

第151行调用函数readLogLines开始读取前面打开的日志设备的日志记录。接下来，我们就分析这些日志记录的读取过程。

4.6.3 日志记录的读取过程

从前面4.6.2小节的内容可以知道，Logcat工具是从源代码文件logcat.cpp中的函数readLogLines开始读取日志记录的，我们分段来阅读这个函数的实现。

system/core/logcat/logcat.cpp

```
01 static void readLogLines(log_device_t* devices)
02 {
03     log_device_t* dev;
04     int max = 0;
05     int ret;
```

```
06      int queued_lines = 0;
07      bool sleep = true;
08
09      int result;
10      fd_set readset;
11
12      for (dev=devices; dev; dev = dev->next) {
13          if (dev->fd > max) {
14              max = dev->fd;
15          }
16      }
17
18      while (1) {
19          do {
20              timeval timeout = { 0, 5000 /* 5ms */ }; // If we oversleep it's ok, i.e. ignore
EINTR.
21              FD_ZERO(&readset);
22              for (dev=devices; dev; dev = dev->next) {
23                  FD_SET(dev->fd, &readset);
24              }
25              result = select(max + 1, &readset, NULL, NULL, sleep ? NULL : &timeout);
26          } while (result == -1 && errno == EINTR);
```

由于Logcat工具有可能同时打开了多个日志设备，因此，第19行到第26行的while循环就使用函数select来同时监控它们是否有内容可读，即是否有新的日志记录需要读取。调用函数select时，需要指定所监控的日志设备文件描述符的最大值，因此，第12行到第16行的for循环就用来查找这些打开的日志设备中的最大文件描述符，并保存在变量max中。在调用函数select之前，第22行到第24行的for循环把所有打开的日志设备的文件描述符都保存到一个fd_set对象readset中，接着第25行就调用函数select来监控前面所打开的日志设备是否有新的日志记录可读，其中，指定的等待时间为5毫秒，即如果在5毫秒之内，所有打开的日志设备都没有新的日志记录可读，那么函数select就超时返回，即它的返回值为0；否则，函数select就会将fd_set对象readset中的相应位设置为1，表示该位所对应的日志设备有新的日志记录可读，这时候函数select的返回值是大于0的。如果在调用函数select的过程中，Logcat工具有信号需要处理，那么函数select的返回值就会等于-1，并且错误代码errno等于EINTR，表示Logcat工具需要重新调用函数select来检查打开的日志设备是否有新的日志记录可读。

当函数跳出第19行到26行的while循环之后，有可能是等待超时，也有可能是所监控的日志设备中有新的日志记录可读，因此，我们需要分两种情况来分析日志记录的读取过程。

首先分析日志设备中有新的日志记录可读的情况，如下所示。

system/core/logcat/logcat.cpp

```
27          if (result >= 0) {
28              for (dev=devices; dev; dev = dev->next) {
29                  if (FD_ISSET(dev->fd, &readset)) {
30                      queued_entry_t* entry = new queued_entry_t();
31                      /* NOTE: driver guarantees we read exactly one full entry */
32                      ret = read(dev->fd, entry->buf, LOGGER_ENTRY_MAX_LEN);
33                      if (ret < 0) {
34                          if (errno == EINTR) {
35                              delete entry;
36                              goto next;
37                          }
38                          if (errno == EAGAIN) {
39                              delete entry;
40                              break;
41                          }
42                          perror("logcat read");
43                          exit(EXIT_FAILURE);
44                      }
```

```
45                  else if (!ret) {
46                      fprintf(stderr, "read: Unexpected EOF!\n");
47                      exit(EXIT_FAILURE);
48                  }
49
50                  entry->entry.msg[entry->entry.len] = '\0';
51
52                  dev->enqueue(entry);
53                  ++queued_lines;
54              }
55          }
```

第28行到第55行的for循环依次处理有新的日志记录可读的日志设备。如果一个日志设备有新的日志记录可读，那么第29行的if语句就会为true，接着第30行就会分配一个queued_entry_t结构体entry，并且第32行调用函数read把该日志设备中的一条新的日志记录读到结构体entry内部的缓冲区buf中。如果在读取日志记录的过程中出现错误，那么Logcat工具就会调用函数exit直接退出。但是如果错误码等于EINTR或者EAGAIN，就需要特殊处理。如果错误代码errno等于EINTR，就说明Logcat工具在读取日志记录的过程中被信号打断，因此，Logcat工具会重新执行next标签处的代码，即重新执行第18行的while循环来监控所打开的日志设备中是否有新的日志记录可读。如果错误码等于EAGAIN，就说明该日志设备在打开时指定了O_NONBLOCK标志，即以非阻塞的模式来打开该日志设备，这时候Logcat工具就会跳出第28行的for循环，继续往下执行。

如果第32行成功地从相应的日志设备中读取到新的日志记录，那么第52行就会将它加入到相应的日志设备的日志记录队列中，并且第53行就会将队列中的日志记录计数queued_lines增加1，表示Logcat工具当前正在等待显示的日志记录条数。

第28行的for循环执行完成之后，就从每个有新的日志记录的日志设备中读出一条日志记录。

注意

这些日志设备中的可读日志记录数可能不只一条，因此，接下来还需要执行第18行的while循环来继续读取这些日志设备中的其他日志记录。不过，在继续读取这些剩余的日志记录之前，Logcat工具先处理前面已经从日志设备中读取出来的日志记录，如下所示。

system/core/logcat/logcat.cpp

```
56              if (result == 0) {
57                  // we did our short timeout trick and there's nothing new
58                  // print everything we have and wait for more data
59                  sleep = true;
60                  while (true) {
61                      chooseFirst(devices, &dev);
62                      if (dev == NULL) {
63                          break;
64                      }
65                      if (g_tail_lines == 0 || queued_lines <= g_tail_lines) {
66                          printNextEntry(dev);
67                      } else {
68                          skipNextEntry(dev);
69                      }
70                      --queued_lines;
71                  }
72
73                  // the caller requested to just dump the log and exit
74                  if (g_nonblock) {
75                      exit(0);
76                  }
77              } else {
78                  // print all that aren't the last in their list
```

```
79                  sleep = false;
80                  while (g_tail_lines == 0 || queued_lines > g_tail_lines) {
81                      chooseFirst(devices, &dev);
82                      if (dev == NULL || dev->queue->next == NULL) {
83                          break;
84                      }
85                      if (g_tail_lines == 0) {
86                          printNextEntry(dev);
87                      } else {
88                          skipNextEntry(dev);
89                      }
90                      --queued_lines;
91                  }
92              }
93          }
94 next:
95          ;
96      }
97  }
```

第56行到第92行的if语句块是用来处理日志记录输出的，主要通过chooseFirst、printNextEntry和skipNextEntry三个函数来实现。

由于Logcat工具是按照写入时间的先后顺序来输出日志记录的，因此，在输出已经读取的日志记录之前，Logcat工具首先会调用函数chooseFirst找到包含有最早的未输出日志记录的日志设备，它的实现如下所示。

system/core/logcat/logcat.cpp

```
1 static void chooseFirst(log_device_t* dev, log_device_t** firstdev) {
2     for (*firstdev = NULL; dev != NULL; dev = dev->next) {
3         if (dev->queue != NULL && (*firstdev == NULL || cmp(dev->queue, (*firstdev)->queue) < 0)) {
4             *firstdev = dev;
5         }
6     }
7 }
```

因为每一个日志设备的日志队列都是按照写入时间的先后顺序来排列日志记录的，因此，函数chooseFirst只要比较日志队列中的第一个日志记录的写入时间，就可以找到包含有最早的未输出日志记录的日志设备。

真正用来输出日志记录的函数是printNextEntry，它的实现如下所示。

system/core/logcat/logcat.cpp

```
1 static void printNextEntry(log_device_t* dev) {
2     maybePrintStart(dev);
3     if (g_printBinary) {
4         printBinary(&dev->queue->entry);
5     } else {
6         processBuffer(dev, &dev->queue->entry);
7     }
8     skipNextEntry(dev);
9 }
```

第2行调用函数maybePrintStart来检查日志设备dev中的日志记录是否是第一次输出。如果是，就会首先输出一行提示性文字，如下所示。

system/core/logcat/logcat.cpp

```
01 static void maybePrintStart(log_device_t* dev) {
02     if (!dev->printed) {
03         dev->printed = true;
04         if (g_devCount > 1 && !g_printBinary) {
05             char buf[1024];
```

```
06              snprintf(buf, sizeof(buf), "--------- beginning of %s\n", dev->device);
07              if (write(g_outFD, buf, strlen(buf)) < 0) {
08                  perror("output error");
09                  exit(-1);
10              }
11          }
12      }
13  }
```

回到函数printNextEntry中，如果在启动Logcat工具时，指定了B选项，那么全局变量g_printBinary的值就会被设置为1，表示Logcat工具要以二进制格式来输出读取到的日志记录，因此，第4行就会调用函数printBinary来输出已经读取到的日志记录；否则，第6行就会调用函数processBuffer来输出已经读取到的日志记录。在接下来的4.6.4小节中分析日志记录的输出过程时，我们再详细分析这两个函数的实现。

日志队列中的日志记录输出之后，就要将它从队列中删除，这是通过调用函数skipNextEntry来实现的，如下所示。

system/core/logcat/logcat.cpp

```
1 static void skipNextEntry(log_device_t* dev) {
2     maybePrintStart(dev);
3     queued_entry_t* entry = dev->queue;
4     dev->queue = entry->next;
5     delete entry;
6 }
```

回到函数readLogLines中，我们接着分析第56行到第92行的if语句块是如何处理那些已经从日志设备中读取出来的日志记录的。

如果变量result的值等于0，即第56行的if语句为true，就说明前面在调用函数select监控日志设备中是否有新的日志记录可读时超时了。既然所有打开的日志设备都没有新的日志记录可读，第60行到第71行的while循环就是时候输出之前已经读取出来的日志记录了。第61行首先调用函数chooseFirst来获得包含有最早的未输出日志记录的日志设备，然后再考虑是否要输出这条最早的日志记录。如果在启动Logcat工具时，指定了t选项，即限定了可以输出的最新日志记录的条数，那么就需要计算所有日志设备中的未输出日志记录的条数。如果未输出的日志记录条数大于可以输出的最大值，即全局变量g_tail_lines的值，那么就需要将最早的一部分日志记录丢弃；如果没有限定可以输出的最新日志记录的条数，即全局变量g_tail_lines的值为0，那么Logcat工具就会将所有未输出的日志记录输出。处理完成日志设备中的已读取日志记录之后，第74行检查Logcat工具是否以非阻塞的模式来打开日志设备。如果是，由于这时候函数select是超时返回，即日志设备中没有新的日志记录可读，那么Logcat工具就直接从第75行退出了。

如果变量result的值大于0，即第56行的if语句为false，那么Logcat工具就会执行第77行到第92行代码来处理日志设备中的已读取日志记录。在这种情况下，日志设备中可能还有新的日志记录等待读取，因此，它的处理方式就会与函数select超时的情况有所不同。这时候如果没有限定可以输出的最新日志记录的条数，即全局变量g_tail_lines的值为0，那么Logcat工具就不用考虑日志设备中是否还有剩余的日志记录未读取了，它可以立即输出那些已经读取的日志记录；如果限定了可以输出的最新日志记录的条数，即全局变量g_tail_lines的值大于0，那么就要把最早的一部分日志记录删除，直到它们的数量小于等于限定的可以输出的最新日志记录的条数为止。在这种情况下，还需要继续读取日志设备中的其余未读取日志记录，直到所有打开的日志设备都没有新的日志记录可读时，Logcat工具才会将已经读取的日志记录输出。

以上就是日志记录的读取过程。接下来，我们继续分析日志记录的输出过程。

4.6.4 日志记录的输出过程

从前面4.6.3小节的内容可以知道，Logcat工具是通过调用源代码文件logcat.cpp中的函数printNextEntry来输出日志记录的。

在前面的4.6.2小节中还提到，如果全局变量g_printBinary的值等于1，那么就说明以二进制格式来输出日志记录。这时候函数printNextEntry就调用函数printBinary来处理。

system/core/logcat/logcat.cpp

```
1 void printBinary(struct logger_entry *buf)
2 {
3     size_t size = sizeof(logger_entry) + buf->len;
4     int ret;
5
6     do {
7         ret = write(g_outFD, buf, size);
8     } while (ret < 0 && errno == EINTR);
9 }
```

由于不需要对日志记录进行解析，即不用将它的优先级、标签以及内容解析出来，因此，函数printBinary的实现很简单，可直接调用函数write将它输出到文件或者打印到标准输出中。

如果全局变量g_printBinary的值等于0，那么就要以文本格式来输出日志记录。这时候函数printNextEntry就调用函数processBuffer来处理。

system/core/logcat/logcat.cpp

```
01 static void processBuffer(log_device_t* dev, struct logger_entry *buf)
02 {
03     int bytesWritten = 0;
04     int err;
05     AndroidLogEntry entry;
06     char binaryMsgBuf[1024];
07
08     if (dev->binary) {
09         err = android_log_processBinaryLogBuffer(buf, &entry, g_eventTagMap,
10                 binaryMsgBuf, sizeof(binaryMsgBuf));
11         //printf(">>> pri=%d len=%d msg='%s'\n",
12         //    entry.priority, entry.messageLen, entry.message);
13     } else {
14         err = android_log_processLogBuffer(buf, &entry);
15     }
16     if (err < 0) {
17         goto error;
18     }
19
20     if (android_log_shouldPrintLine(g_logformat, entry.tag, entry.priority)) {
21         if (false && g_devCount > 1) {
22             binaryMsgBuf[0] = dev->label;
23             binaryMsgBuf[1] = ' ';
24             bytesWritten = write(g_outFD, binaryMsgBuf, 2);
25             if (bytesWritten < 0) {
26                 perror("output error");
27                 exit(-1);
28             }
29         }
30
31         bytesWritten = android_log_printLogLine(g_logformat, g_outFD, &entry);
32
33         if (bytesWritten < 0) {
34             perror("output error");
```

```
35              exit(-1);
36          }
37      }
38
39      g_outByteCount += bytesWritten;
40
41      if (g_logRotateSizeKBytes > 0
42          && (g_outByteCount / 1024) >= g_logRotateSizeKBytes
43      ) {
44          rotateLogs();
45      }
46
47 error:
48      //fprintf (stderr, "Error processing record\n");
49      return;
50 }
```

前面得到的日志记录只是一块二进制数据，保存在一个logger_entry结构体中，因此，函数processBuffer在将它输出之前，首先要将它的内容转换为一个AndroidLogEntry结构体。AndroidLogEntry结构体描述了一条日志记录的写入时间、优先级、标签、内容以及写入进程ID。如果日志记录的类型是二进制格式的，即是类型为events的日志记录，那么函数processBuffer就会调用函数android_log_processBinaryLogBuffer对它进行解析；否则，就调用函数android_log_processLogBuffer对它进行解析。日志记录解析完成之后，函数processBuffer最后就调用函数android_log_printLogLine将它输出到文件或者打印到标准输出中。这三个函数的实现我们在后面再详细分析，现在先完成对函数processBuffer的分析。

一条日志记录解析完成之后，Logcat工具就得到了它的优先级和标签。由于在启动Logcat工具时，可能设置了日志记录输出过滤器，因此，函数processBuffer就需要调用函数android_log_shouldPrintLine判断一条日志记录是否能够输出。在前面的4.6.2小节中提到，Logcat工具的日志记录输出过滤器列表保存在全局变量g_logformat中，因此，函数processBuffer就以它作为参数来调用函数android_log_shouldPrintLine，判断一条日志记录是否能够输出，它的实现如下所示。

system/core/liblog/logprint.c

```
1 /**
2  * returns 1 if this log line should be printed based on its priority
3  * and tag, and 0 if it should not
4  */
5 int android_log_shouldPrintLine (
6         AndroidLogFormat *p_format, const char *tag, android_LogPriority pri)
7 {
8     return pri >= filterPriForTag(p_format, tag);
9 }
```

函数filterPriForTag首先在参数p_format中检查是否对日志记录标签tag设置了输出过滤器。如果设置了，那么函数filterPriForTag就会返回它的过滤优先级。只有这个过滤优先级低于即将要输出的日志记录的优先级时，该日志记录才可以输出。

函数filterPriForTag的实现如下所示。

system/core/liblog/logprint.c

```
01 static android_LogPriority filterPriForTag(
02         AndroidLogFormat *p_format, const char *tag)
03 {
04     FilterInfo *p_curFilter;
05
06     for (p_curFilter = p_format->filters
07             ; p_curFilter != NULL
08             ; p_curFilter = p_curFilter->p_next
```

```
09     ) {
10         if (0 == strcmp(tag, p_curFilter->mTag)) {
11             if (p_curFilter->mPri == ANDROID_LOG_DEFAULT) {
12                 return p_format->global_pri;
13             } else {
14                 return p_curFilter->mPri;
15             }
16         }
17     }
18
19     return p_format->global_pri;
20 }
```

第6行到第17行的for循环检查在参数p_format的日志记录输出过滤器列表中是否为日志记录标签tag设置了输出过滤器。如果设置了，就返回该日志记录标签所对应的优先级；否则，就返回全局设置的日志记录输出优先级，即返回参数p_format的成员变量global_pri。另外，如果我们为日志记录标签tag设置了输出过滤器，但是将该输出过滤器的优先级设置为ANDROID_LOG_DEFAULT，那么函数filterPriForTag实际上返回的是全局设置的日志记录输出优先级，如第12行代码所示。

回到函数processBuffer中，如果函数android_log_shouldPrintLine的返回值为true，那么它就会先调用函数android_log_printLogLine输出日志记录，然后再将输出的日志记录的字节数bytesWritten增加到全局变量g_outByteCount中，表示到目前为止，一共往文件g_outFD中输出了多少个字节的日志记录。一旦输出到文件g_outFD中的日志记录的字节数大于全局变量g_logRotateSizeKBytes的值时，那么Logcat就会将接下来的其他日志记录输出到另外一个文件中。全局变量g_logRotateSizeKBytes的值是由Logcat工具的启动选项r来指定的，如果没有指定该选项，那么全局变量g_logRotateSizeKBytes的值就会等于0，表示将所有的日志记录都输出到同一个文件中。如果全局变量g_logRotateSizeKBytes的值大于0，那么总共可以用来作为日志记录输出文件的个数就由Logcat工具的选项n来指定。如果没有指定选项n，那么默认就有四个日志记录输出文件。第44行调用函数rotateLogs来设置下一个日志记录输出文件，它的实现如下所示。

system/core/logcat/logcat.cpp

```
01 static void rotateLogs()
02 {
03     int err;
04
05     // Can't rotate logs if we're not outputting to a file
06     if (g_outputFileName == NULL) {
07         return;
08     }
09
10     close(g_outFD);
11
12     for (int i = g_maxRotatedLogs ; i > 0 ; i--) {
13         char *file0, *file1;
14
15         asprintf(&file1, "%s.%d", g_outputFileName, i);
16
17         if (i - 1 == 0) {
18             asprintf(&file0, "%s", g_outputFileName);
19         } else {
20             asprintf(&file0, "%s.%d", g_outputFileName, i - 1);
21         }
22
23         err = rename (file0, file1);
24
25         if (err < 0 && errno != ENOENT) {
26             perror("while rotating log files");
```

```
27          }
28
29          free(file1);
30          free(file0);
31      }
32
33      g_outFD = openLogFile (g_outputFileName);
34
35      if (g_outFD < 0) {
36          perror ("couldn't open output file");
37          exit(-1);
38      }
39
40      g_outByteCount = 0;
41
42  }
```

假设在启动Logcat工具时，通过选项f来指定日志记录输出文件为“logfile”，并且通过选项n来指定日志记录输出文件个数为3，那么函数rotateLogs就分别将日志记录输出文件设置为logfile、logfile.1、logfile.2和logfile.3。

注意

这四个文件是循环使用的，即当文件logfile.3的大小达到限定的值之后，Logcat工具就将接下来的日志记录重新输出到文件logfile、logfile.1和logfile.2中。依此类推，当文件logfile.2的大小再次达到限定的值之后，最后又将新的日志记录输出到文件logfile.3中。

分析完成函数processBuffer的实现之后，接下来我们开始分析函数android_log_processLogBuffer、android_log_processBinaryLogBuffer和android_log_printLogLine的实现。

首先分析函数android_log_processLogBuffer的实现，它是用来解析类型为main、system和radio的日志记录的，如下所示。

system/core/liblog/logprint.c

```
01 /**
02  * Splits a wire-format buffer into an AndroidLogEntry
03  * entry allocated by caller. Pointers will point directly into buf
04  *
05  * Returns 0 on success and -1 on invalid wire format (entry will be
06  * in unspecified state)
07  */
08 int android_log_processLogBuffer(struct logger_entry *buf,
09                                  AndroidLogEntry *entry)
10 {
11     size_t tag_len;
12
13     entry->tv_sec = buf->sec;
14     entry->tv_nsec = buf->nsec;
15     entry->priority = buf->msg[0];
16     entry->pid = buf->pid;
17     entry->tid = buf->tid;
18     entry->tag = buf->msg + 1;
19     tag_len = strlen(entry->tag);
20     entry->messageLen = buf->len - tag_len - 3;
21     entry->message = entry->tag + tag_len + 1;
22
23     return 0;
24 }
```

日志记录的优先级字段的长度为1个字节，保存在logger_entry 结构体buf内部的缓冲区msg的第一

个字节中。日志记录的标签字段从logger_entry 结构体buf内部的缓冲区msg的第二个字节开始，一直到后面的‘\0’字符为止，因此，第19行可以通过调用函数strlen来计算它的长度。logger_entry 结构体buf内部的缓冲区msg 剩余的字节即为日志记录的内容字段，它的长度等于缓冲区的总长度减去日志记录标签的长度，再减去1个字节的日志记录优先级，以及2个字符串结束字符‘\0’，其中一个是日志记录标签字段的，另一个是日志记录内容字段的。

接下来，我们继续分析函数android_log_processBinaryLogBuffer的实现，它是用来解析类型为events的日志记录的，我们分段来阅读。

system/core/liblog/logprint.c

```
01 /**
02  * Convert a binary log entry to ASCII form.
03  *
04  * For convenience we mimic the processLogBuffer API.  There is no
05  * pre-defined output length for the binary data, since we're free to format
06  * it however we choose, which means we can't really use a fixed-size buffer
07  * here.
08  */
09 int android_log_processBinaryLogBuffer(struct logger_entry *buf,
10     AndroidLogEntry *entry, const EventTagMap* map, char* messageBuf,
11     int messageBufLen)
12 {
13     size_t inCount;
14     unsigned int tagIndex;
15     const unsigned char* eventData;
16
17     entry->tv_sec = buf->sec;
18     entry->tv_nsec = buf->nsec;
19     entry->priority = ANDROID_LOG_INFO;
20     entry->pid = buf->pid;
21     entry->tid = buf->tid;
22
23     /*
24      * Pull the tag out.
25      */
26     eventData = (const unsigned char*) buf->msg;
27     inCount = buf->len;
28     if (inCount < 4)
29         return -1;
30     tagIndex = get4LE(eventData);
31     eventData += 4;
32     inCount -= 4;
33
34     if (map != NULL) {
35         entry->tag = android_lookupEventTag(map, tagIndex);
36     } else {
37         entry->tag = NULL;
38     }
```

第17行到21行代码用来初始化AndroidLogEntry 结构体entry中的日志记录写入时间、写入进程ID以及优先级等信息。从前面4.1小节中的图4–3可以知道，类型为events的日志记录是没有优先级这个字段的，但是为了和其他类型的日志记录统一处理，Logcat工具将它们的优先级设置为ANDROID_LOG_INFO。

第26行得到logger_entry 结构体buf内部的缓冲区msg的地址，并且保存在变量eventData中。由于类型为events的日志记录的标签是使用一个整数值来描述的，并且保存在logger_entry 结构体buf内部的缓冲区msg的前面4个字节中，因此第30行就调用函数get4LE将它获取回来，并且保存在变量tagIndex中。

函数get4LE的实现如下所示。

system/core/liblog/logprint.c

```
1 /*
2  * Extract a 4-byte value from a byte stream.
3  */
4 static inline uint32_t get4LE(const uint8_t* src)
5 {
6     return src[0] | (src[1] << 8) | (src[2] << 16) | (src[3] << 24);
7 }
```

它只是简单地将缓冲区src前面4个字节的内容组合在一起形成一个整数，然后返回给调用者。

回到函数android_log_processBinaryLogBuffer中，接下来第34行检查参数map的值是否为NULL。如果是，就直接将AndroidLogEntry 结构体entry的成员变量tag设置为NULL，表示当前正在处理的一个日志记录的标签值没有对应的描述字符串；否则，第35行就会调用函数android_lookupEventTag在参数map中找到与日志记录标签值tagIndex对应的描述字符串。

注意

参数map指向一个EventTagMap结构体，它是Logcat工具在启动时，通过解析目标设备上的/system/etc/event-log-tags文件得到的。因此，Logcat工具就可以通过它来找到与日志记录标签值tagIndex对应的描述字符串。

函数android_lookupEventTag的实现如下所示。

system/core/liblog/ event_tag_map.c

```
01 /*
02  * Look up an entry in the map.
03  *
04  * The entries are sorted by tag number, so we can do a binary search.
05  */
06 const char* android_lookupEventTag(const EventTagMap* map, int tag)
07 {
08     int hi, lo, mid;
09
10     lo = 0;
11     hi = map->numTags-1;
12
13     while (lo <= hi) {
14         int cmp;
15
16         mid = (lo+hi)/2;
17         cmp = map->tagArray[mid].tagIndex - tag;
18         if (cmp < 0) {
19             /* tag is bigger */
20             lo = mid + 1;
21         } else if (cmp > 0) {
22             /* tag is smaller */
23             hi = mid - 1;
24         } else {
25             /* found */
26             return map->tagArray[mid].tagStr;
27         }
28     }
29
30     return NULL;
31 }
```

在前面的4.6.2小节中介绍Logcat工具的初始化过程时提到，Logcat工具解析完成目标设备上的/system/etc/event-log-tags文之后，就得到了一个日志记录标签描述表，表中描述了与每一个日志标签所对应的描述字符串，最后会按照标签值从小到大的顺序保存在EventTagMap结构体内部的EventTag

结构体数组tagArray中。因此，函数android_lookupEventTag就可以使用二分法从这个数组中得到与日志记录标签值tag对应的描述字符串，并且将它返回给调用者。

回到函数android_log_processBinaryLogBuffer中，继续往下执行。

system/core/liblog/logprint.c

```
39      /*
40       * If we don't have a map, or didn't find the tag number in the map,
41       * stuff a generated tag value into the start of the output buffer and
42       * shift the buffer pointers down.
43       */
44      if (entry->tag == NULL) {
45          int tagLen;
46
47          tagLen = snprintf(messageBuf, messageBufLen, "[%d]", tagIndex);
48          entry->tag = messageBuf;
49          messageBuf += tagLen+1;
50          messageBufLen -= tagLen+1;
51      }
52
53      /*
54       * Format the event log data into the buffer.
55       */
56      char* outBuf = messageBuf;
57      size_t outRemaining = messageBufLen-1;      /* leave one for nul byte */
58      int result;
59      result = android_log_printBinaryEvent(&eventData, &inCount, &outBuf,
60                  &outRemaining);
```

如果前面没有在EventTagMap结构体map中找到即将要输出的日志记录的标签值描述字符串，即第44行的if语句为true，那么第47行就将该标签值作为要输出的日志记录的标签值描述字符串。接着第59行调用函数android_log_printBinaryEvent继续解析日志记录的内容字段。

要输出的日志记录解析完成之后，输出结果就保存在缓冲区messageBuf中。在调用函数android_log_printBinaryEvent来解析要输出的日志记录的内容字段之前，第57行会在缓冲区messageBuf的后面保留一个字节，用来保存一个字符串结束字符‘\0’。这样，Logcat工具就可以将缓冲区messageBuf的内容作为一个字符串输出到文件或者打印到标准输出中。

函数android_log_printBinaryEvent的实现如下所示。

system/core/liblog/logprint.c

```
001 /*
002  * Recursively convert binary log data to printable form.
003  *
004  * This needs to be recursive because you can have lists of lists.
005  *
006  * If we run out of room, we stop processing immediately.  It's important
007  * for us to check for space on every output element to avoid producing
008  * garbled output.
009  *
010  * Returns 0 on success, 1 on buffer full, -1 on failure.
011  */
012 static int android_log_printBinaryEvent(const unsigned char** pEventData,
013     size_t* pEventDataLen, char** pOutBuf, size_t* pOutBufLen)
014 {
015     const unsigned char* eventData = *pEventData;
016     size_t eventDataLen = *pEventDataLen;
017     char* outBuf = *pOutBuf;
018     size_t outBufLen = *pOutBufLen;
019     unsigned char type;
020     size_t outCount;
021     int result = 0;
```

```
022
023     if (eventDataLen < 1)
024         return -1;
025     type = *eventData++;
026     eventDataLen--;
027
028     //fprintf(stderr, "--- type=%d (rem len=%d)\n", type, eventDataLen);
029
030     switch (type) {
031     case EVENT_TYPE_INT:
032         /* 32-bit signed int */
033         {
034             int ival;
035
036             if (eventDataLen < 4)
037                 return -1;
038             ival = get4LE(eventData);
039             eventData += 4;
040             eventDataLen -= 4;
041
042             outCount = snprintf(outBuf, outBufLen, "%d", ival);
043             if (outCount < outBufLen) {
044                 outBuf += outCount;
045                 outBufLen -= outCount;
046             } else {
047                 /* halt output */
048                 goto no_room;
049             }
050         }
051         break;
052     case EVENT_TYPE_LONG:
053         /* 64-bit signed long */
054         {
055             long long lval;
056
057             if (eventDataLen < 8)
058                 return -1;
059             lval = get8LE(eventData);
060             eventData += 8;
061             eventDataLen -= 8;
062
063             outCount = snprintf(outBuf, outBufLen, "%lld", lval);
064             if (outCount < outBufLen) {
065                 outBuf += outCount;
066                 outBufLen -= outCount;
067             } else {
068                 /* halt output */
069                 goto no_room;
070             }
071         }
072         break;
073     case EVENT_TYPE_STRING:
074         /* UTF-8 chars, not NULL-terminated */
075         {
076             unsigned int strLen;
077
078             if (eventDataLen < 4)
079                 return -1;
080             strLen = get4LE(eventData);
081             eventData += 4;
082             eventDataLen -= 4;
083
084             if (eventDataLen < strLen)
```

```
085                 return -1;
086
087             if (strLen < outBufLen) {
088                 memcpy(outBuf, eventData, strLen);
089                 outBuf += strLen;
090                 outBufLen -= strLen;
091             } else if (outBufLen > 0) {
092                 /* copy what we can */
093                 memcpy(outBuf, eventData, outBufLen);
094                 outBuf += outBufLen;
095                 outBufLen -= outBufLen;
096                 goto no_room;
097             }
098             eventData += strLen;
099             eventDataLen -= strLen;
100             break;
101         }
102     case EVENT_TYPE_LIST:
103         /* N items, all different types */
104         {
105             unsigned char count;
106             int i;
107
108             if (eventDataLen < 1)
109                 return -1;
110
111             count = *eventData++;
112             eventDataLen--;
113
114             if (outBufLen > 0) {
115                 *outBuf++ = '[';
116                 outBufLen--;
117             } else {
118                 goto no_room;
119             }
120
121             for (i = 0; i < count; i++) {
122                 result = android_log_printBinaryEvent(&eventData, &eventDataLen,
123                         &outBuf, &outBufLen);
124                 if (result != 0)
125                     goto bail;
126
127                 if (i < count-1) {
128                     if (outBufLen > 0) {
129                         *outBuf++ = ',';
130                         outBufLen--;
131                     } else {
132                         goto no_room;
133                     }
134                 }
135             }
136
137             if (outBufLen > 0) {
138                 *outBuf++ = ']';
139                 outBufLen--;
140             } else {
141                 goto no_room;
142             }
143         }
144         break;
145     default:
146         fprintf(stderr, "Unknown binary event type %d\n", type);
147         return -1;
```

```
148     }
149
150 bail:
151     *pEventData = eventData;
152     *pEventDataLen = eventDataLen;
153     *pOutBuf = outBuf;
154     *pOutBufLen = outBufLen;
155     return result;
156
157 no_room:
158     result = 1;
159     goto bail;
160 }
```

函数android_log_printBinaryEvent正好是在前面4.5小节中介绍的四个日志记录写入函数android_util_EventLog_writeEvent_Integer、android_util_EventLog_writeEvent_Long、android_util_EventLog_writeEvent_String和android_util_EventLog_writeEvent_Array的逆操作，因此，读者可以对照这四个函数来分析函数android_log_printBinaryEvent的实现。简单来说，如果日志记录内容只有一个值，那么函数android_log_printBinaryEvent就会根据它的类型（整数、长整数和字符串）将它从输入缓冲区pEventData取回来，并且保存到输出缓冲区outBuf中。当日志记录的内容是一个列表时，函数android_log_printBinaryEvent就会通过递归调用自己来依次遍历列表中的数据。如果得到的数据是一个整数、长整数或者字符串，那么处理方式就与前面一样；否则，当得到的数据又是一个列表时，函数android_log_printBinaryEvent就会继续对这个子列表执行同样的遍历操作。

函数android_log_printBinaryEvent执行完成之后，如果返回值等于0，那么就说明日志记录解析成功；否则，就说明解析过程中出现了问题，其中，返回值等于-1表示日志记录缓冲区有误，返回值等于1表示输出缓冲区outBuf没有足够的空间来容纳日志记录的内容。

回到函数android_log_processBinaryLogBuffer中，继续往下执行。

system/core/liblog/logprint.c

```
61      if (result < 0) {
62          fprintf(stderr, "Binary log entry conversion failed\n");
63          return -1;
64      } else if (result == 1) {
65          if (outBuf > messageBuf) {
66              /* leave an indicator */
67              *(outBuf-1) = '!';
68          } else {
69              /* no room to output anything at all */
70              *outBuf++ = '!';
71              outRemaining--;
72          }
73          /* pretend we ate all the data */
74          inCount = 0;
75      }
76
77      /* eat the silly terminating '\n' */
78      if (inCount == 1 && *eventData == '\n') {
79          eventData++;
80          inCount--;
81      }
82
83      if (inCount != 0) {
84          fprintf(stderr,
85              "Warning: leftover binary log data (%d bytes)\n", inCount);
86      }
87
88      /*
89       * Terminate the buffer.  The NUL byte does not count as part of
```

```
90      * entry->messageLen.
91      */
92     *outBuf = '\0';
93     entry->messageLen = outBuf - messageBuf;
94     assert(entry->messageLen == (messageBufLen-1) - outRemaining);
95
96     entry->message = messageBuf;
97
98     return 0;
99 }
```

变量result的值是调用函数android_log_printBinaryEvent得到的返回值。如果它的值小于0，即第61行的if语句为true，就说明日志记录内容有误，因此，函数android_log_processBinaryLogBuffer就不用进一步对它进行处理了，第63行直接返回错误码-1；如果它的值等于1，即第64行的if语句为true，那么就说明日志记录输出缓冲区messageBuf空间不足，这时候函数android_log_processBinaryLogBuffer就在缓冲区messageBuf的最后一个字节上写入一个符号‘!’来说明此种情况。

变量inCount表示原始的日志记录缓冲区中还有多少个字节未处理。如果它的值等于1，并且剩余未处理的字符为‘\n’，那么就说明要输出的是一条正常的日志记录，因为每一条类型为events的日志记录总是以‘\n’字符结束的。如果要输出的日志记录不是这种情况，那么第84行和第85行就会输出一条警告信息来说明此种情况。

前面提到，函数android_log_processBinaryLogBuffer在调用函数android_log_printBinaryEvent来解析要输出的日志记录的内容字段之前，会在日志记录输出缓冲区messageBuf的后面保留一个字节，这个字节就是用来写入一个字符串结束符号‘\0’的，如第92行代码所示。这样，Logcat工具就可以把缓冲区messageBuf的内容当作一个字符串来使用，即可以直接将它以文本的形式输出到文件或者打印到标准输出中。

最后，我们分析函数android_log_printLogLine的实现，它是日志记录输出过程中的最后一个步骤，它的实现如下所示。

system/core/liblog/logprint.c

```
01 /**
02  * Either print or do not print log line, based on filter
03  *
04  * Returns count bytes written
05  */
06
07 int android_log_printLogLine(
08     AndroidLogFormat *p_format,
09     int fd,
10     const AndroidLogEntry *entry)
11 {
12     int ret;
13     char defaultBuffer[512];
14     char *outBuffer = NULL;
15     size_t totalLen;
16
17     outBuffer = android_log_formatLogLine(p_format, defaultBuffer,
18             sizeof(defaultBuffer), entry, &totalLen);
19
20     if (!outBuffer)
21         return -1;
22
23     do {
24         ret = write(fd, outBuffer, totalLen);
25     } while (ret < 0 && errno == EINTR);
26
27     if (ret < 0) {
28         fprintf(stderr, "+++ LOG: write failed (errno=%d)\n", errno);
```

```
29          ret = 0;
30          goto done;
31      }
32
33      if (((size_t)ret) < totalLen) {
34          fprintf(stderr, "+++ LOG: write partial (%d of %d)\n", ret,
35                  (int)totalLen);
36          goto done;
37      }
38
39  done:
40      if (outBuffer != defaultBuffer) {
41          free(outBuffer);
42      }
43
44      return ret;
45  }
```

第17行调用函数android_log_formatLogLine来格式化要输出的日志记录，并且将最后结果保存在缓冲区defaultBuffer中。函数android_log_formatLogLine的实现比较简单，它将要输出的日志记录的内容格式化为“<PREFIX> +MESSAGE+<SUFFIX>”的形式。其中，<PREFIX>和<SUFFIX>的内容取决于Logcat工具在初始化时所设置的日志记录输出格式，即参数p_format的内容；而MESSAGE描述了日志记录的内容字段。有了输出结果缓冲区defaultBuffer之后，第23行到第25行的while循环就调用函数write将它输出到文件描述符fd所描述的目标文件中。

在前面的4.6.2小节中介绍Logcat工具的初始化过程时提到，如果在启动Logcat时，指定了选项v，那么就可以指定日志记录的输出格式。选项v后面所跟的参数可以为“brief”、“process”、“tag”、“thread”、“raw”、“time”、“threadtime”或者“long”。每一个参数都对应一个AndroidLogPrintFormat枚举值，分别为FORMAT_BRIEF、FORMAT_PROCESS、FORMAT_TAG、FORMAT_THREAD、FORMAT_RAW、FORMAT_TIME、FORMAT_THREADTIME和FORMAT_LONG。

下面我们就介绍每一个AndroidLogPrintFormat枚举值所对应的日志记录输出格式。

（1）FORMAT_BRIEF：<PREFIX>和<SUFFIX>的内容分别为“<priority>/<tag>(<pid>): ”和“\n”。

（2）FORMAT_PROCESS：<PREFIX>和<SUFFIX>的内容分别为“<priority>(<pid>) ”和“ (<tag>)\n”。

（3）FORMAT_TAG：<PREFIX>和<SUFFIX>的内容分别为“<priority>/(<tag>): ”和“\n”。

（4）FORMAT_THREAD：<PREFIX>和<SUFFIX>的内容分别为“<priority>(<pid>:<tid>) ”和“\n”。

（5）FORMAT_RAW：<PREFIX>和<SUFFIX>的内容分别为空值和“\n”。

（6）FORMAT_TIME：<PREFIX>和<SUFFIX>的内容分别为“<sec>.<nsec> <priority>/<tag>(<pid>): ”和“\n”。

（7）FORMAT_THREADTIME：<PREFIX>和<SUFFIX>的内容分别为 “<sec>.<nsec> <pid> <tid> <priority> <tag>: ”和“\n”。

（8）FORMAT_LONG：<PREFIX>和<SUFFIX>的内容分别为“[<sec>.<nsec> <pid>:<tid> <priority>/<tag>]”和“\n\n”。

例如，假设应用程序（进程ID为396）中调用Java接口android.util.Log来往Logger日志驱动程序中写入了以下一条日志记录：

```
android.util.Log.i("LogTag", "Log Content.");
```

并且假设Logcat工具在启动时，被指定以FORMAT_BRIEF格式来输出该日志记录的内容，那么Logcat工具从Logger日志驱动程序中读出这条日志记录之后，就会打印出以下的输出：

```
I/LogTag(396): Log Content.
```

至此，日志记录的输出过程就分析完成了。

第5章 Binder进程间通信系统

Android应用程序是由Activity、Service、Broadcast Receiver和Content Provider四种类型的组件构成的，它们有可能运行在同一个进程中，也有可能运行在不同的进程中。此外，各种系统组件也运行在独立的进程中，例如，Activity管理服务ActivityManagerService和Package管理服务PackageManagerService都运行在系统进程System中。那么，这些运行在不同进程中的应用程序组件和系统组件是如何通信的呢？

我们知道，Android系统是基于Linux内核开发的。Linux内核提供了丰富的进程间通信机制，如管道（Pipe）、信号（Signal）、消息队列（Message）、共享内存（Share Memory）和插口（Socket）等。然而，Android系统并没有采用这些传统的进程间通信机制，而是开发了一套新的进程间通信机制——Binder。与传统的进程间通信机制相比，Binder进程间通信机制在进程间传输数据时，只需要执行一次复制操作，因此，它不仅提高了效率，而且节省了内存空间[1]。

Binder进程间通信机制是在OpenBinder的基础上实现的[2]，它采用CS通信方式，其中，提供服务的进程称为Server进程，而访问服务的进程称为Client进程。同一个Server进程可以同时运行多个组件来向Client进程提供服务，这些组件称为Service组件[3]。同时，同一个Client进程也可以同时向多个Service组件请求服务，每一个请求都对应有一个Client组件，或者称为Service代理对象。Binder进程间通信机制的每一个Server进程和Client进程都维护一个Binder线程池来处理进程间的通信请求，因此，Server进程和Client进程可以并发地提供和访问服务。Server进程和Client进程的通信要依靠运行在内核空间的Binder驱动程序来进行。Binder驱动程序向用户空间暴露了一个设备文件/dev/binder，使得应用程序进程可以间接地通过它来建立通信通道。Service组件在启动时，会将自己注册到一个Service Manager组件中，以便Client组件可以通过Service Manager组件找到它。因此，我们将Service Manager组件称为Binder进程间通信机制的上下文管理者，同时由于它也需要与普通的Server进程和Client进程通信，我们也将它看作是一个Service组件，只不过它是一个特殊的Service组件。

以上描述的Binder进程间通信机制中涉及了Client、Service、Service Manager和Binder驱动程序四个角色，它们的关系如图5-1所示。

1 使用共享内存在进程间传输数据，虽然也只需要执行一次复制操作，但是它一般要结合其他的进程间通信机制来同步信息。

2 OpenBinder是一种进程间通信机制，它最初是由Be公司开发的，后来由Palm公司接手开发和维护，最后Google对其进行改造，并且应用在Android系统中，具体可以参考http://en.wikipedia.org/wiki/OpenBinder。

3 注意：Binder进程间通信机制中的Service组件与Android应用程序中的Service组件是两个不同的概念，在后面的第8章中，我们再详细分析Android应用程序组件Service的实现原理。

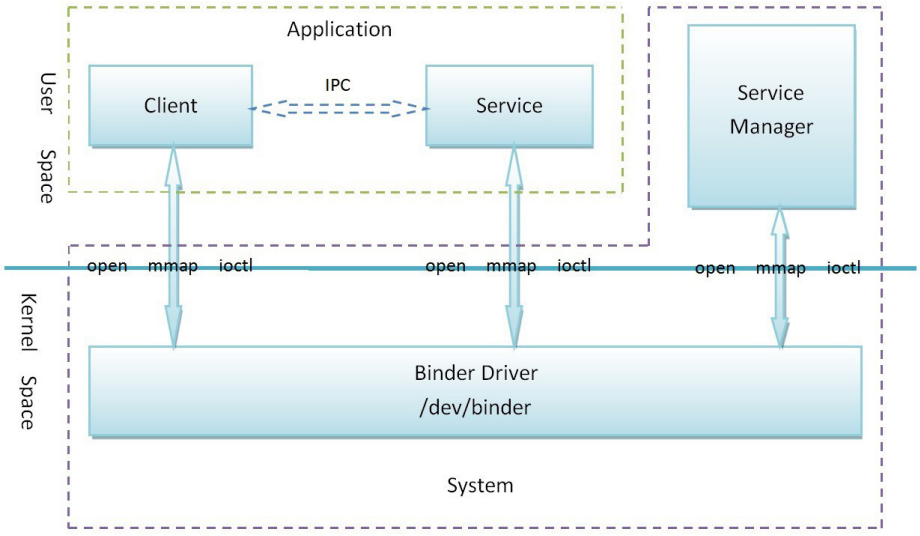


图5-1　Binder进程间通信机制

从图5-1可以看出，Client、Service和Service Manager运行在用户空间，而Binder驱动程序运行在内核空间，其中，Service Manager和Binder驱动程序由系统负责提供，而Client和Service组件由应用程序来实现。Client、Service和Service Manager均是通过系统调用open、mmap和ioctl来访问设备文件/dev/binder，从而实现与Binder驱动程序的交互，而交互的目的就是为了能够间接地执行进程间通信。

在本章内容中，我们首先分析Binder驱动程序的基础知识，然后介绍它在用户空间中的一个Binder进程间通信库。了解了这些知识点之后，再通过一个实例说明应用程序进程是如何使用Binder进程间通信库来通信的。接着我们再继续进一步分析Binder驱动程序的实现原理，包括Binder对象的引用计数技术和死亡通知机制。

在Android系统中，Binder进程间通信机制的使用是相当广泛的，因此，掌握Binder进程间通信机制，对理解整个Android系统的实现是有非常大的帮助的。为了进一步加深对Binder进程间通信机制的理解，我们分析了Binder进程间通信机制的四个使用情景，分别是：

（1）Service Manager的启动过程。

（2）Service Manager代理对象的获取过程。

（3）Service组件的启动过程。

（4）Service代理对象的获取过程。

上述四个使用情景都是基于Binder进程间通信机制的C/C++实现来分析的。由于Android应用程序是使用Java语言开发的，因此，在本章的最后一节中，我们将详细介绍Binder进程间通信机制的Java调用接口。

5.1　Binder驱动程序

Binder驱动程序实现在内核空间中，它的目录结构如下：

```
~/Android/kernel/goldfish
----drivers
     ----staging
          ----android
               ----binder.h
               ----binder.c
```

它主要由binder.h和binder.c两个源文件组成。下面我们就开始介绍Binder驱动程序的基础知识，包括基础数据结构、初始化过程，以及设备文件/dev/binder的打开（open）、内存映射（mmap）和内核缓冲区管理等操作。

5.1.1 基础数据结构

在Binder驱动程序中有两种类型的数据结构，其中一种是在内部使用的，另一种是在内部和外部均会使用到的。接下来，我们分别对这些数据结构进行介绍。

struct binder_work

kernel/goldfish/drivers/staging/android/binder.c

```
01 struct binder_work {
02         struct list_head entry;
03         enum {
04                 BINDER_WORK_TRANSACTION = 1,
05                 BINDER_WORK_TRANSACTION_COMPLETE,
06                 BINDER_WORK_NODE,
07                 BINDER_WORK_DEAD_BINDER,
08                 BINDER_WORK_DEAD_BINDER_AND_CLEAR,
09                 BINDER_WORK_CLEAR_DEATH_NOTIFICATION,
10         } type;
11 };
```

结构体binder_work用来描述待处理的工作项，这些工作项有可能属于一个进程，也有可能属于一个进程中的某一个线程。成员变量entry用来将该结构体嵌入到一个宿主结构中，成员变量type用来描述工作项的类型。根据成员变量type的取值，Binder 驱动程序就可以判断出一个binder_work结构体嵌入到什么类型的宿主结构中。

struct binder_node

kernel/goldfish/drivers/staging/android/binder.c

```
01 struct binder_node {
02         int debug_id;
03         struct binder_work work;
04         union {
05                 struct rb_node rb_node;
06                 struct hlist_node dead_node;
07         };
08         struct binder_proc *proc;
09         struct hlist_head refs;
10         int internal_strong_refs;
11         int local_weak_refs;
12         int local_strong_refs;
13         void __user *ptr;
14         void __user *cookie;
15         unsigned has_strong_ref : 1;
16         unsigned pending_strong_ref : 1;
17         unsigned has_weak_ref : 1;
18         unsigned pending_weak_ref : 1;
19         unsigned has_async_transaction : 1;
20         unsigned accept_fds : 1;
21         int min_priority : 8;
22         struct list_head async_todo;
23 };
```

结构体binder_node用来描述一个Binder实体对象。每一个Service组件在Binder驱动程序中都对应有一个Binder实体对象，用来描述它在内核中的状态。Binder驱动程序通过强引用计数和弱引用计数

技术来维护它们的生命周期。

成员变量proc指向一个Binder实体对象的宿主进程。在Binder驱动程序中，这些宿主进程通过一个binder_proc结构体来描述。宿主进程使用一个红黑树来维护它内部所有的Binder实体对象，而每一个Binder实体对象的成员变量rb_node就正好是这个红黑树中的一个节点。如果一个Binder实体对象的宿主进程已经死亡了，那么这个Binder实体对象就会通过它的成员变量dead_node保存在一个全局的hash列表中。

由于一个Binder实体对象可能会同时被多个Client组件引用，因此，Binder驱动程序就使用结构体binder_ref来描述这些引用关系，并且将引用了同一个Binder实体对象的所有引用都保存在一个hash列表中。这个hash列表通过Binder实体对象的成员变量refs来描述，而Binder驱动程序通过这个成员变量就可以知道有哪些Client组件引用了同一个Binder实体对象。

成员变量internal_strong_refs和local_strong_refs均是用来描述一个Binder实体对象的强引用计数，而成员变量local_weak_refs用来描述一个Binder实体对象的弱引用计数。当一个Binder实体对象请求一个Service组件来执行某一个操作时，会增加该Service组件的强引用计数或者弱引用计数，相应地，Binder实体对象会将其成员变量has_strong_ref和has_weak_ref的值设置为1。当一个Service组件完成一个Binder实体对象所请求的操作之后，Binder实体对象就会请求减少该Service组件的强用计数或者弱引用计数。Binder实体对象在请求一个Service组件增加或者减少强引用计数或者弱引用计数的过程中，会将其成员变量pending_strong_ref或者pending_weak_ref的值设置为1；而当该Service组件增加或者减少了强引用计数或者弱引用计数之后，Binder实体对象就会将这两个成员变量的值设置为0。

成员变量ptr和cookie分别指向一个用户空间地址，它们用来描述用户空间中的一个Service组件，其中，成员变量cookie指向该Service组件的地址，而成员变量ptr指向该Service组件内部的一个引用计数对象（类型为weakref_impl）的地址。

成员变量has_async_transaction用来描述一个Binder实体对象是否正在处理一个异步事务。如果是，它的值就等于1，否则等于0。一般情况下，Binder驱动程序都是将一个事务保存在一个线程的一个todo队列中的，表示要由该线程来处理该事务。每一个事务都关联着一个Binder实体对象，表示该事务的目标处理对象，即要求与该Binder实体对象对应的Service组件在指定的线程中处理该事务。然而，当Binder驱动程序发现一个事务是异步事务时，就会将它保存在目标Binder实体对象的一个异步事务队列中，这个异步事务队列就是由该目标Binder实体对象的成员变量async_todo来描述的。异步事务的定义是那些单向的进程间通信请求，即不需要等待应答的进程间通信请求，与此相对的便是同步事务。因为不需要等待应答，Binder驱动程序就认为异步事务的优先级低于同步事务，具体就表现为在同一时刻，一个Binder实体对象的所有异步事务至多只有一个会得到处理，其余的都等待在异步事务队列中，而同步事务就没有这个限制。

当一个Binder实体对象的引用计数由0变成1，或者由1变成0时，Binder驱动程序就会请求相应的Service组件增加或者减少其引用计数。这时候Binder驱动程序就会将该引用计数修改操作封装成一个类型为binder_node的工作项，即将一个Binder实体对象的成员变量work的值设置为BINDER_WORK_NODE，并且将它添加到相应进程的todo队列中去等待处理。

成员变量min_priority表示一个Binder实体对象在处理一个来自Client进程的请求时，它所要求的处理线程，即Server进程中的一个线程，应该具备的最小线程优先级，这样就保证了与该Binder实体对象对应的Service组件可以在一个具有一定优先级的线程中处理一个来自Client进程的通信请求。一个Service组件在将自己注册到Binder驱动程序时，可以指定这个最小线程优先级，而Binder驱动程序会把这个最小线程优先级保存在相应的Binder实体对象的成员变量min_priority中。

成员变量accept_fds用来描述一个Binder实体对象是否可以接收包含有文件描述符的进程间通信数据。如果它的值等于1，就表示可以接收；否则，就表示禁止接收。当一个进程向另外一个进程发送

的数据中包含有文件描述符时，Binder驱动程序就会自动在目标进程中打开一个相同的文件。基于安全性考虑，Binder驱动程序就要通过成员变量accept_fds来防止源进程在目标进程中打开文件。

最后，成员变量debug_id用来标志一个Binder实体对象的身份，它是用来帮助调试Binder驱动程序的。

struct binder_ref_death

kernel/goldfish/drivers/staging/android/binder.c

```
1 struct binder_ref_death {
2         struct binder_work work;
3         void __user *cookie;
4 };
```

结构体binder_ref_death用来描述一个Service组件的死亡接收通知。在正常情况下，一个Service组件被其他Client进程引用时，它是不可以销毁的。然而，Client进程是无法控制它所引用的Service组件的生命周期的，因为Service组件所在的进程可能会意外地崩溃，从而导致它意外地死亡。一个折中的处理办法是，Client进程能够在它所引用的Service组件死亡时获得通知，以便可以做出相应的处理。这时候Client进程就需要将一个用来接收死亡通知的对象的地址注册到Binder驱动程序中。

成员变量cookie用来保存负责接收死亡通知的对象的地址，成员变量work的取值为BINDER_WORK_DEAD_BINDER、BINDER_WORK_CLEAR_DEATH_NOTIFICATION或者BINDER_WORK_DEAD_BINDER_AND_CLEAR，用来标志一个具体的死亡通知类型。

Binder驱动程序决定要向一个Client进程发送一个Service组件死亡通知时，会将一个binder_ref_death结构体封装成一个工作项，并且根据实际情况来设置该结构体的成员变量work的值，最后将这个工作项添加到Client进程的todo队列中去等待处理。

在下面两种情况下，Binder驱动程序会向一个Client进程发送一个Service组件的死亡通知。

（1）当Binder驱动程序监测到一个Service组件死亡时，它就会找到该Service组件对应的Binder实体对象，然后通过Binder实体对象的成员变量refs就可以找到所有引用了它的Client进程，最后就找到这些Client进程所注册的死亡接收通知，即一个binder_ref_death结构体。这时候Binder驱动程序就会将该binder_ref_death结构体添加到Client进程的todo队列中去等待处理。在这种情况下，Binder驱动程序将死亡通知的类型设置为BINDER_WORK_DEAD_BINDER。

（2）当Client进程向Binder驱动程序注册一个死亡接收通知时，如果它所引用的Service组件已经死亡，那么Binder驱动程序就会马上发送一个死亡通知给该Client进程。在这种情况下，Binder驱动程序也会将死亡通知的类型设置为BINDER_WORK_DEAD_BINDER。

另外，当Client进程向Binder驱动程序注销一个死亡接收通知时，Binder驱动程也会向该Client进程的todo队列发送一个类型为binder_ref_death的工作项，用来表示注销结果。这时候又需要分两种情况来考虑。

（1）如果Client进程在注销一个死亡接收通知时，相应的Service组件还没有死亡，那么Binder驱动程序就会找到之前所注册的一个binder_ref_death结构体，并且将它的类型work修改为BINDER_WORK_CLEAR_DEATH_NOTIFICATION，然后再将该binder_ref_death结构体封装成一个工作项添加到该Client进程的todo队列中去等待处理。

（2）如果Client进程在注销一个死亡接收通知时，相应的Service组件已经死亡，那么Binder驱动程序就会找到之前所注册的一个binder_ref_death结构体，并且将它的类型work修改为BINDER_WORK_DEAD_BINDER_AND_CLEAR，然后再将该binder_ref_death结构体封装成一个工作项添加到该Client进程的todo队列中去等待处理。

Client进程在处理这个工作项时，通过对应的binder_ref_death结构体的成员变量work就可以区分注销结果了，即它所引用的Service组件是否已经死亡。

struct binder_ref

kernel/goldfish/drivers/staging/android/binder.c

```
01 struct binder_ref {
02         /* Lookups needed: */
03         /*   node + proc => ref (transaction) */
04         /*   desc + proc => ref (transaction, inc/dec ref) */
05         /*   node => refs + procs (proc exit) */
06         int debug_id;
07         struct rb_node rb_node_desc;
08         struct rb_node rb_node_node;
09         struct hlist_node node_entry;
10         struct binder_proc *proc;
11         struct binder_node *node;
12         uint32_t desc;
13         int strong;
14         int weak;
15         struct binder_ref_death *death;
16 };
```

结构体binder_ref用来描述一个Binder引用对象。每一个Client组件在Binder驱动程序中都对应有一个Binder引用对象，用来描述它在内核中的状态。Binder驱动程序通过强引用计数和弱引用计数技术来维护它们的生命周期。

成员变量node用来描述一个Binder引用对象所引用的Binder实体对象。前面在介绍结构体binder_node时提到，每一个Binder实体对象都有一个hash列表，用来保存那些引用了它的Binder引用对象，而这些Binder引用对象的成员变量node_entry正好是这个hash列表的节点。

成员变量desc是一个句柄值，或者称为描述符，它是用来描述一个Binder引用对象的。在Client进程的用户空间中，一个Binder引用对象是使用一个句柄值来描述的，因此，当Client进程的用户空间通过Binder驱动程序来访问一个Service组件时，它只需要指定一个句柄值，Binder驱动程序就可以通过该句柄值找到对应的Binder引用对象，然后再根据该Binder引用对象的成员变量node找到对应的Binder实体对象，最后就可以通过该Binder实体对象找到要访问的Service组件。

注意

一个Binder引用对象的句柄值在进程范围内是唯一的，因此，在两个不同的进程中，同一个句柄值可能代表的是两个不同的目标Service组件。

成员变量proc指向一个Binder引用对象的宿主进程。一个宿主进程使用两个红黑树来保存它内部所有的Binder引用对象，它们分别以句柄值和对应的Binder实体对象的地址来作为关键字保存这些Binder引用对象，而这些Binder引用对象的成员变量rb_node_desc和rb_node_node就正好是这两个红黑树中的节点。

成员变量strong和weak分别用来描述一个Binder引用对象的强引用计数和弱引用计数，Binder驱动程序正是通过它们来维护一个Binder引用对象的生命周期的。

成员变量death指向一个Service组件的死亡接收通知。当Client进程向Binder驱动程序注册一个它所引用的Service组件的死亡接收通知时，Binder驱动程序就会创建一个binder_ref_death结构体，然后保存在对应的Binder引用对象的成员变量death中。

最后，成员变量debug_id用来标志一个Binder引用对象的身份，它是用来帮助调试Binder驱动程序的。

struct binder_buffer

kernel/goldfish/drivers/staging/android/binder.c

```
01 struct binder_buffer {
02         struct list_head entry; /* free and allocated entries by addesss */
```

```
03          struct rb_node rb_node; /* free entry by size or allocated entry */
04                                  /* by address */
05          unsigned free : 1;
06          unsigned allow_user_free : 1;
07          unsigned async_transaction : 1;
08          unsigned debug_id : 29;
09
10          struct binder_transaction *transaction;
11
12          struct binder_node *target_node;
13          size_t data_size;
14          size_t offsets_size;
15          uint8_t data[0];
16 };
```

结构体binder_buffer用来描述一个内核缓冲区，它是用来在进程间传输数据的。每一个使用Binder进程间通信机制的进程在Binder驱动程序中都有一个内核缓冲区列表，用来保存Binder驱动程序为它所分配的内核缓冲区，而成员变量entry正好是这个内核缓冲区列表的一个节点。同时，进程又使用了两个红黑树来分别保存那些正在使用的内核缓冲区，以及空闲的内核缓冲区。如果一个内核缓冲区是空闲的，即它的成员变量free的值等于1，那么成员变量rb_node就是空闲内核缓冲区红黑树中的一个节点；否则，成员变量rb_node就是正在使用内核缓冲区红黑树中的一个节点。

成员变量transaction 和target_node用来描述一个内核缓冲区正在交给哪一个事务以及哪一个Binder实体对象使用。Binder驱动程序使用一个binder_transaction结构体来描述一个事务，每一个事务都关联有一个目标Binder实体对象。Binder驱动程序将事务数据保存在一个内核缓冲区中，然后将它交给目标Binder实体对象处理，而目标Binder实体对象再将该内核缓冲区的内容交给相应的Service组件处理。Service组件处理完成该事务之后，如果发现传递给它的内核缓冲区的成员变量allow_user_free的值为1，那么该Service组件就会请求Binder驱动程序释放该内核缓冲区。

如果与一个内核缓冲区关联的是一个异步事务，那么Binder驱动程序就会将该内核缓冲区的成员变量async_transaction的值设置为1；否则，就将它的值设置为0。Binder驱动程序限制了分配给异步事务的内核缓冲区的大小，这样做的目的是为了保证同步事务可以优先得到内核缓冲区，以便可以快速地对该同步事务进行处理。

成员变量data指向一块大小可变的数据缓冲区，它是真正用来保存通信数据的。数据缓冲区保存的数据划分为两种类型，其中一种是普通数据，另一种是Binder对象。Binder驱动程序不关心数据缓冲区中的普通数据，但是必须要知道里面的Binder对象，因为它需要根据它们来维护内核中的Binder实体对象和Binder引用对象的生命周期。例如，如果数据缓冲区中包含了一个Binder引用，并且该数据缓冲区是传递给另外一个进程的，那么Binder驱动程序就需要为另外一个进程创建一个Binder引用对象，并且增加相应的Binder实体对象的引用计数，因为它也被另外的这个进程引用了。由于数据缓冲区中的普通数据和Binder对象是混合在一起保存的，它们之间并没有固定的顺序，因此，Binder驱动程序就需要额外的数据来找到里面的Binder对象。在数据缓冲区的后面，有一个偏移数组，它记录了数据缓冲区中每一个Binder对象在数据缓冲区中的位置。偏移数组的大小保存在成员变量offsets_size中，而数据缓冲区的大小保存在成员变量data_size中。

最后，成员变量debug_id用来标志一个内核缓冲区的身份，它是用来帮助调试Binder驱动程序的。

struct binder_proc

kernel/goldfish/drivers/staging/android/binder.c

```
01 struct binder_proc {
02          struct hlist_node proc_node;
03          struct rb_root threads;
04          struct rb_root nodes;
```

```
05          struct rb_root refs_by_desc;
06          struct rb_root refs_by_node;
07          int pid;
08          struct vm_area_struct *vma;
09          struct task_struct *tsk;
10          struct files_struct *files;
11          struct hlist_node deferred_work_node;
12          int deferred_work;
13          void *buffer;
14          ptrdiff_t user_buffer_offset;
15
16          struct list_head buffers;
17          struct rb_root free_buffers;
18          struct rb_root allocated_buffers;
19          size_t free_async_space;
20
21          struct page **pages;
22          size_t buffer_size;
23          uint32_t buffer_free;
24          struct list_head todo;
25          wait_queue_head_t wait;
26          struct binder_stats stats;
27          struct list_head delivered_death;
28          int max_threads;
29          int requested_threads;
30          int requested_threads_started;
31          int ready_threads;
32          long default_priority;
33 };
```

结构体binder_proc用来描述一个正在使用Binder进程间通信机制的进程。当一个进程调用函数open来打开设备文件/dev/binder时，Binder驱动程序就会为它创建一个binder_proc结构体，并且将它保存在一个全局的hash列表中，而成员变量proc_node就正好是该hash列表中的一个节点。此外，成员变量pid、tsk和files分别指向了进程的进程组ID、任务控制块和打开文件结构体数组。

进程打开了设备文件/dev/binder之后，还必须调用函数mmap将它映射到进程的地址空间来，实际上是请求Binder驱动程序为它分配一块内核缓冲区，以便可以用来在进程间传输数据。Binder驱动程序为进程分配的内核缓冲区的大小保存在成员变量buffer_size中。这些内核缓冲区有两个地址，其中一个是内核空间地址，另外一个是用户空间地址。内核空间地址是在Binder驱动程序内部使用的，保存在成员变量buffer中，而用户空间地址是在应用程序进程内部使用的，保存在成员变量vma中。这两个地址相差一个固定的值，保存在成员变量user_buffer_offset中。这样，给定一个用户空间地址或者一个内核空间地址，Binder驱动程序就可以计算出另外一个地址的大小。

注意

这两个地址指的都是虚拟地址，它们对应的物理页面保存在成员变量pages中。成员变量pages是类型为struct page*的一个数组，数组中的每一个元素都指向一个物理页面。Binder驱动程序一开始时只为该内核缓冲区分配一个物理页面，后面不够使用时，再继续分配。

成员变量buffer指向的是一块大的内核缓冲区，Binder驱动程序为了方便对它进行管理，会将它划分成若干个小块。这些小块的内核缓冲区就是使用前面所介绍的结构体binder_buffer来描述的，它们保存在一个列表中，按照地址值从小到大的顺序来排列。成员变量buffers指向的便是该列表的头部。列表中的小块内核缓冲区有的是正在使用的，即已经分配了物理页面；有的是空闲的，即还没有分配物理页面，它们分别组织在两个红黑树中，其中，前者保存在成员变量allocated_buffers所描述的红黑树中，而后者保存在成员变量free_buffers所描述的红黑树中。此外，成员变量buffer_free保存了空闲内

核缓冲区的大小，而成员变量free_async_space保存了当前可以用来保存异步事务数据的内核缓冲区的大小。

前面提到，每一个使用了Binder进程间通信机制的进程都有一个Binder线程池，用来处理进程间通信请求，这个Binder线程池是由Binder驱动程序来维护的。结构体binder_proc的成员变量threads是一个红黑树的根节点，它以线程ID作为关键字来组织一个进程的Binder线程池。进程可以调用函数ioctl将一个线程注册到Binder驱动程序中，同时，当进程没有足够的空闲线程在处理进程间通信请求时，Binder驱动程序也可以主动要求进程注册更多的线程到Binder线程池中。Binder驱动程序最多可以主动请求进程注册的线程的数量保存在成员变量max_threads中，而成员变量ready_threads表示进程当前的空闲Binder线程数目。

注意

成员变量max_threads并不是表示Binder线程池中的最大线程数目，进程本身可以主动注册任意数目的线程到Binder线程池中。Binder驱动程序每一次主动请求进程注册一个线程时，都会将成员变量requested_threads的值加1；而当进程响应这个请求之后，Binder驱动程序就会将成员变量requested_threads的值减1，而且将成员变量requested_threads_started的值加1，表示Binder驱动程序已经主动请求进程注册了多少个线程到Binder线程池中。

当进程接收到一个进程间通信请求时，Binder驱动程序就将该请求封装成一个工作项，并且加入到进程的待处理工作项队列中，这个队列使用成员变量todo来描述。Binder线程池中的空闲Binder线程会睡眠在由成员变量wait所描述的一个等待队列中，当它们的宿主进程的待处理工作项队列增加了新的工作项之后，Binder驱动程序就会唤醒这些线程，以便它们可以去处理新的工作项。成员变量default_priority的值被初始化为进程的优先级。当一个线程处理一个工作项时，它的线程优先级有可能会被设置为其宿主进程的优先级，即设置为成员变量default_priority的值，这是由于线程是代表其宿主进程来处理一个工作项的。线程在处理一个工作项时的优先级还会受到其他因素的影响，后面我们再详细描述。

一个进程内部包含了一系列的Binder实体对象和Binder引用对象，进程使用三个红黑树来组织它们，其中，成员变量nodes所描述的红黑树是用来组织Binder实体对象的，它以Binder实体对象的成员变量ptr作为关键字；而成员变量refs_by_desc和refs_by_node所描述的红黑树是用来组织Binder引用对象的，前者以Binder引用对象的成员变量desc作为关键字，而后者以Binder引用对象的成员变量node作为关键字。

成员变量deferred_work_node是一个hash列表，用来保存进程可以延迟执行的工作项。这些延迟工作项有三种类型，如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
1 enum {
2         BINDER_DEFERRED_PUT_FILES    = 0x01,
3         BINDER_DEFERRED_FLUSH        = 0x02,
4         BINDER_DEFERRED_RELEASE      = 0x04,
5 };
```

Binder驱动程序为进程分配内核缓冲区时，会为这个内核缓冲区创建一个文件描述符，进程可以通过这个文件描述符将该内核缓冲区映射到自己的地址空间。当进程不再需要使用Binder进程间通信机制时，它就会通知Binder驱动程序关闭该文件描述符，并且释放之前所分配的内核缓冲区。由于这不是一个马上就需要完成的操作，因此，Binder驱动程序就会创建一个BINDER_DEFERRED_PUT_FILES类型的工作项来延迟执行该操作。

前面提到，Binder线程池中的空闲Binder线程是睡眠在一个等待队列中的，进程可以通过调用函

数flush来唤醒这些线程，以便它们可以检查进程是否有新的工作项需要处理。这时候Binder驱动程序就会创建一个BINDER_DEFERRED_FLUSH类型的工作项，以便可以延迟执行唤醒空闲Binder线程的操作。

当进程不再使用Binder进程间通信机制时，它就会调用函数close来关闭设备文件/dev/binder，这时候Binder驱动程序就会释放之前为它分配的资源，例如，释放进程结构体binder_proc、Binder实体对象结构体binder_node以及Binder引用对象结构体binder_ref等。由于资源的释放操作是一个比较耗时的操作，因此，Binder驱动程序会创建一个BINDER_DEFERRED_RELEASE类型的事务来延迟执行它们。

Binder驱动程序将所有的延迟执行的工作项保存在一个hash列表中。如果一个进程有延迟执行的工作项，那么成员变量deferred_work_node就刚好是该hash列表中的一个节点，并且使用成员变量deferred_work来描述该延迟工作项的具体类型。

当一个进程所引用的Service组件死亡时，Binder驱动程序就会向该进程发送一个死亡通知。这个正在发出的死亡通知被封装成一个类型为BINDER_WORK_DEAD_BINDER或者BINDER_WORK_DEAD_BINDER_AND_CLEAR的工作项，并且保存在由成员变量delivered_death所描述的一个队列中，表示Binder驱动程序正在向进程发送的死亡通知。当进程接收到这个死亡通知之后，它便会通知Binder驱动程序，这时候Binder驱动程序就会将对应的工作项从成员变量delivered_death所描述的队列中删除。

最后，成员变量stats是用来统计进程数据的，例如，进程接收到的进程间通信请求的次数。

struct binder_thread

kernel/goldfish/drivers/staging/android/binder.c

```
01 struct binder_thread {
02         struct binder_proc *proc;
03         struct rb_node rb_node;
04         int pid;
05         int looper;
06         struct binder_transaction *transaction_stack;
07         struct list_head todo;
08         uint32_t return_error; /* Write failed, return error code in read buf */
09         uint32_t return_error2; /* Write failed, return error code in read */
10                 /* buffer. Used when sending a reply to a dead process that */
11                 /* we are also waiting on */
12         wait_queue_head_t wait;
13         struct binder_stats stats;
14 };
```

结构体binder_thread用来描述Binder线程池中的一个线程，其中，成员变量proc指向其宿主进程。前面在介绍进程结构体binder_proc时提到，进程结构体binder_proc使用一个红黑树来组织其Binder线程池中的线程，其中，结构体binder_thread的成员变量rb_node就是该红黑树中的一个节点。

一个Binder线程的ID和状态是通过成员变量pid和looper来描述的。线程状态的取值如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
1 enum {
2         BINDER_LOOPER_STATE_REGISTERED    = 0x01,
3         BINDER_LOOPER_STATE_ENTERED       = 0x02,
4         BINDER_LOOPER_STATE_EXITED        = 0x04,
5         BINDER_LOOPER_STATE_INVALID       = 0x08,
6         BINDER_LOOPER_STATE_WAITING       = 0x10,
7         BINDER_LOOPER_STATE_NEED_RETURN   = 0x20
8 };
```

一个线程注册到Binder驱动程序时，Binder驱动程序就会为它创建一个binder_thread结构体，并且将它的状态初始化为BINDER_LOOPER_STATE_NEED_RETURN，表示该线程需要马上返回到用户空

间。由于一个线程在注册为Binder线程时可能还没有准备好去处理进程间通信请求，因此，最好返回到用户空间去做准备工作。此外，当进程调用函数flush来刷新它的Binder线程池时，Binder线程池中的线程的状态也会被重置为BINDER_LOOPER_STATE_NEED_RETURN。

一个线程注册到Binder驱动程序之后，它接着就会通过BC_REGISTER_LOOPER或者BC_ENTER_LOOPER协议来通知Binder驱动程序，它可以处理进程间通信请求了，这时候Binder驱动程序就会将它的状态设置为BINDER_LOOPER_STATE_REGISTERED或者BINDER_LOOPER_STATE_ENTERED。如果一个线程是应用程序主动注册的，那么它就通过BC_ENTER_LOOPER协议来通知Binder驱动程序，它已经准备就绪处理进程间通信请求了；如果一个线程是Binder驱动程序请求创建的，那么它就通过BC_REGISTER_LOOPER协议来通知Binder驱动程序，这时候Binder驱动程序就会增加它所请求进程创建的Binder线程的数目。

当一个Binder线程处于空闲状态时，Binder驱动程序就会把它的状态设置为BINDER_LOOPER_STATE_WAITING；而当一个Binder线程退出时，它会通过BC_EXIT_LOOPER协议来通知Binder驱动程序，这时候Binder驱动程序就会将它的状态设置为BINDER_LOOPER_STATE_EXITED。在异常情况下，一个Binder线程的状态会被设置为BINDER_LOOPER_STATE_INVALID，例如，当该线程已经处于BINDER_LOOPER_STATE_REGISTERED状态时，如果它又再次通过BC_ENTER_LOOPER协议来通知Binder驱动程序它已经准备就绪了，那么Binder驱动程序就会将它的状态设置为BINDER_LOOPER_STATE_INVALID。

当一个来自Client进程的请求指定要由某一个Binder线程来处理时，这个请求就会加入到相应的binder_thread结构体的成员变量todo所表示的队列中，并且会唤醒这个线程来处理，因为这时候这个线程可能处于睡眠状态。

当Binder驱动程序决定将一个事务交给一个Binder线程处理时，它就会将该事务封装成一个binder_transaction结构体，并且将它添加到由线程结构体binder_thread的成员变量transaction_stack所描述的一个事务堆栈中。结构体binder_transaction的设计很巧妙，后面我们再详细介绍它的定义。

当一个Binder线程在处理一个事务T1并需要依赖于其他的Binder线程来处理另外一个事务T2时，它就会睡眠在由成员变量wait所描述的一个等待队列中，直到事务T2处理完成为止。

一个Binder线程在处理一个事务时，如果出现了异常情况，那么Binder驱动程序就会将相应的错误码保存在其成员变量return_error和reutrn_error2中，这时候线程就会将这些错误返回给用户空间应用程序处理。

最后，成员变量stats是用来统计Binder线程数据的，例如，Binder线程接收到的进程间通信请求的次数。

struct binder_transaction

kernel/goldfish/drivers/staging/android/binder.c

```
01 struct binder_transaction {
02         int debug_id;
03         struct binder_work work;
04         struct binder_thread *from;
05         struct binder_transaction *from_parent;
06         struct binder_proc *to_proc;
07         struct binder_thread *to_thread;
08         struct binder_transaction *to_parent;
09         unsigned need_reply : 1;
10         /*unsigned is_dead : 1;*/ /* not used at the moment */
11
12         struct binder_buffer *buffer;
13         unsigned int    code;
14         unsigned int    flags;
```

```
15          long    priority;
16          long    saved_priority;
17          uid_t   sender_euid;
18 };
```

结构体binder_transaction用来描述进程间通信过程，这个过程又称为一个事务。成员变量need_reply用来区分一个事务是同步的还是异步的。同步事务需要等待对方回复，这时候它的成员变量need_reply的值就会设置为1；否则就设置为0，表示这是一个异步事务，不需要等待回复。

成员变量from指向发起事务的线程，称为源线程；成员变量to_proc和to_thread分别指向负责处理该事务的进程和线程，称为目标进程和目标线程。当Binder驱动程序为目标进程或者目标线程创建一个事务时，就会将该事务的成员变量work的值设置为BINDER_WORK_TRANSACTION，并且将它添加到目标进程或者目标线程的todo队列中去等待处理。

成员变量priority和sender_euid分别用来描述源线程的优先级和用户ID。通过这两个成员变量，目标进程或者目标线程就可以识别事务发起方的身份。

一个线程在处理一个事务时，Binder驱动程序需要修改它的线程优先级，以便满足源线程和目标Service组件的要求。Binder驱动程序在修改一个线程的优先级之前，会将它原来的线程优先级保存在一个事务结构体的成员变量saved_priority中，以便线程处理完成该事务后可以恢复原来的优先级。前面在介绍结构体binder_node时提到，目标线程在处理一个事务时，它的线程优先级不能低于目标Service组件所要求的线程优先级，而且也不能低于源线程的优先级。这时候Binder驱动程序就会将这二者中的较大值设置为目标线程的优先级。

成员变量buffer指向Binder驱动程序为该事务分配的一块内核缓冲区，它里面保存了进程间通信数据。成员变量code和flags是直接从进程间通信数据中复制过来的，后面在介绍结构体binder_transaction_data时，我们再详细分析。

成员变量from_parent和to_parent分别描述一个事务所依赖的另外一个事务，以及目标线程下一个需要处理的事务。假设线程A发起了一个事务T1，需要由线程B来处理；线程B在处理事务T1时，又需要线程C先处理事务T2；线程C在处理事务T2时，又需要线程A先处理事务T3。这样，事务T1就依赖于事务T2，而事务T2又依赖于事务T3，它们的关系如下：

```
T2->from_parent = T1;
T3->from_parent = T2;
```

对于线程A来说，它需要处理的事务有两个，分别是T1和T3，它首先要处理事务T3，然后才能处理事务T1，因此，事务T1和T3的关系如下：

```
T3->to_parent = T1;
```

考虑这样一个情景：如果线程C在发起事务T3给线程A所属的进程来处理时，Binder驱动程序选择了该进程的另外一个线程D来处理该事务，这时候会出现什么情况呢？这时候线程A就会处于空闲等待状态，什么也不能做，因为它必须要等线程D处理完成事务T3后，它才可以继续执行事务T1。在这种情况下，与其让线程A闲着，还不如把事务T3交给它来处理，这样线程D就可以去处理其他事务，提高了进程的并发性。

现在，关键的问题又来了——Binder驱动程序在分发事务T3给目标进程处理时，它是如何知道线程A属于目标进程，并且正在等待事务T3的处理结果的？当线程C在处理事务T2时，就会将事务T2放在其事务堆栈transaction_stack的最前端。这样当线程C发起事务T3给线程A所属的进程处理时，Binder驱动程序就可以沿着线程C的事务堆栈transaction_stack向下遍历，即沿着事务T2的成员变量from_parent向下遍历，最后就会发现事务T3的目标进程等于事务T1的源进程，并且事务T1是由线程A发起的，这时候它就知道线程A正在等待事务T3的处理结果了。

注意

线程A在处理事务T3时，Binder驱动程序会将事务T3放在其事务堆栈transaction_stack的最前端，而在此之前，该事务堆栈transaction_stack的最前端指向的是事务T1。为了能够让线程A处理完成事务T3之后，接着处理事务T1，Binder驱动程序会将事务T1保存在事务T3的成员变量to_parent中。等到线程A处理完成事务T3之后，就可以通过事务T3的成员变量to_parent找到事务T1，再将它放在线程A的事务堆栈transaction_stack的最前端了。

最后，成员变量debug_id用来标志一个事务结构体的身份，它是用来帮助调试Binder驱动程序的。

以上介绍的结构体都是在Binder驱动程序内部使用的。前面提到，应用程序进程在打开了设备文件/dev/binder之后，需要通过IO控制函数ioctl来进一步与Binder驱动程序进行交互，因此，Binder驱动程序就提供了一系列的IO控制命令来和应用程序进程通信。在这些IO控制命令中，最重要的便是BINDER_WRITE_READ命令了，它的定义如下所示。

kernel/goldfish/drivers/staging/android/binder.h

```
1 #define BINDER_WRITE_READ               _IOWR('b', 1, struct binder_write_read)
```

IO控制命令BINDER_WRITE_READ后面所跟的参数是一个binder_write_read结构体，它的定义如下所示。

kernel/goldfish/drivers/staging/android/binder.h

```
1 struct binder_write_read {
2         signed long     write_size;     /* bytes to write */
3         signed long     write_consumed; /* bytes consumed by driver */
4         unsigned long   write_buffer;
5         signed long     read_size;      /* bytes to read */
6         signed long     read_consumed;  /* bytes consumed by driver */
7         unsigned long   read_buffer;
8 };
```

结构体binder_write_read用来描述进程间通信过程中所传输的数据。这些数据包括输入数据和输出数据，其中，成员变量write_size、write_consumed和write_buffer用来描述输入数据，即从用户空间传输到Binder驱动程序的数据；而成员变量read_size、read_consumed和read_buffer用来描述输出数据，即从Binder驱动程序返回给用户空间的数据，它也是进程间通信结果数据。

成员变量write_buffer指向一个用户空间缓冲区的地址，里面保存的内容即为要传输到Binder驱动程序的数据。缓冲区write_buffer的大小由成员变量write_size来指定，单位是字节。成员变量write_consumed用来描述Binder驱动程序从缓冲区write_buffer中处理了多少个字节的数据。

成员变量read_buffer也是指向一个用户空间缓冲区的地址，里面保存的内容即为Binder驱动程序返回给用户空间的进程间通信结果数据。缓冲区read_buffer的大小由成员变量read_size来指定，单位是字节。成员变量read_consumed用来描述用户空间应用程序从缓冲区read_buffer中处理了多少个字节的数据。

缓冲区write_buffer和read_buffer的数据格式如图5-2所示。



图5-2　缓冲区write_buffer和read_buffer的数据格式

它们都是一个数组，数组的每一个元素都由一个通信协议代码及其通信数据组成。协议代码又分为两种类型，其中一种是在输入缓冲区write_buffer中使用的，称为命令协议代码，另一种是在输出缓冲区read_buffer中使用的，称为返回协议代码。命令协议代码通过BinderDriverCommandProtocol枚举

值来定义，而返回协议代码通过BinderDriverReturnProtocol枚举值来定义。

命令协议代码的定义如下所示。

enum BinderDriverCommandProtocol

kernel/goldfish/drivers/staging/android/binder.h

```
01 enum BinderDriverCommandProtocol {
02         BC_TRANSACTION = _IOW('c', 0, struct binder_transaction_data),
03         BC_REPLY = _IOW('c', 1, struct binder_transaction_data),
04         /*
05          * binder_transaction_data: the sent command.
06          */
07
08         BC_ACQUIRE_RESULT = _IOW('c', 2, int),
09         /*
10          * not currently supported
11          * int:  0 if the last BR_ATTEMPT_ACQUIRE was not successful.
12          * Else you have acquired a primary reference on the object.
13          */
14
15         BC_FREE_BUFFER = _IOW('c', 3, int),
16         /*
17          * void *: ptr to transaction data received on a read
18          */
19
20         BC_INCREFS = _IOW('c', 4, int),
21         BC_ACQUIRE = _IOW('c', 5, int),
22         BC_RELEASE = _IOW('c', 6, int),
23         BC_DECREFS = _IOW('c', 7, int),
24         /*
25          * int: descriptor
26          */
27
28         BC_INCREFS_DONE = _IOW('c', 8, struct binder_ptr_cookie),
29         BC_ACQUIRE_DONE = _IOW('c', 9, struct binder_ptr_cookie),
30         /*
31          * void *: ptr to binder
32          * void *: cookie for binder
33          */
34
35         BC_ATTEMPT_ACQUIRE = _IOW('c', 10, struct binder_pri_desc),
36         /*
37          * not currently supported
38          * int: priority
39          * int: descriptor
40          */
41
42         BC_REGISTER_LOOPER = _IO('c', 11),
43         /*
44          * No parameters.
45          * Register a spawned looper thread with the device.
46          */
47
48         BC_ENTER_LOOPER = _IO('c', 12),
49         BC_EXIT_LOOPER = _IO('c', 13),
50         /*
51          * No parameters.
52          * These two commands are sent as an application-level thread
53          * enters and exits the binder loop, respectively.  They are
54          * used so the binder can have an accurate count of the number
55          * of looping threads it has available.
56          */
57
```

```
58          BC_REQUEST_DEATH_NOTIFICATION = _IOW('c', 14, struct binder_ptr_cookie),
59          /*
60           * void *: ptr to binder
61           * void *: cookie
62           */
63
64          BC_CLEAR_DEATH_NOTIFICATION = _IOW('c', 15, struct binder_ptr_cookie),
65          /*
66           * void *: ptr to binder
67           * void *: cookie
68           */
69
70          BC_DEAD_BINDER_DONE = _IOW('c', 16, void *),
71          /*
72           * void *: cookie
73           */
74  };
```

命令协议代码BC_TRANSACTION和BC_REPLY后面跟的通信数据使用一个结构体binder_transaction_data来描述。当一个进程请求另外一个进程执行某一个操作时，源进程就使用命令协议代码BC_TRANSACTION来请求Binder驱动程序将通信数据传递到目标进程；当目标进程处理完成源进程所请求的操作之后，它就使用命令协议代码BC_REPLY来请求Binder驱动程序将结果数据传递给源进程。

命令协议代码BC_ACQUIRE_RESULT在当前的Binder驱动程序实现中不支持。

命令协议代码BC_FREE_BUFFER后面跟的通信数据是一个整数，它指向了在Binder驱动程序内部所分配的一块内核缓冲区。Binder驱动程序就是通过这个内核缓冲区将源进程的通信数据传递到目标进程的。当目标进程处理完成源进程的通信请求之后，它就会使用命令协议代码BC_FREE_BUFFER来通知Binder驱动程序释放这个内核缓冲区。

命令协议代码BC_INCREFS、BC_ACQUIRE、BC_RELEASE和BC_DECREFS后面跟的通信数据是一个整数，它描述了一个Binder引用对象的句柄值，其中，命令协议代码BC_INCREFS和BC_DECREFS分别用来增加和减少一个Binder引用对象的弱引用计数；而命令协议代码BC_ACQUIRE和BC_RELEASE分别用来增加和减少一个Binder引用对象的强引用计数。

命令协议代码BC_INCREFS_DONE和BC_ACQUIRE_DONE后面跟的通信数据使用一个结构体binder_ptr_cookie来描述。Binder驱动程序第一次增加一个Binder实体对象的强引用计数或者弱引用计数时，就会使用返回协议代码BR_ACQUIRE或者BR_INCREFS来请求对应的Server进程增加对应的Service组件的强引用计数或者弱引用计数。当Server进程处理完成这两个请求之后，就会分别使用命令协议代码BC_INCREFS_DONE和BC_ACQUIRE_DONE将操作结果返回给Binder驱动程序。

命令协议代码BC_ATTEMPT_ACQUIRE在当前的Binder驱动程序实现中不支持。

命令协议代码BC_REGISTER_LOOPER、BC_ENTER_LOOPER和BC_EXIT_LOOPER后面不需要指定通信数据。一方面，当一个线程将自己注册到Binder驱动程序之后，它接着就会使用命令协议代码BC_ENTER_LOOPER来通知Binder驱动程序，它已经准备就绪处理进程间通信请求了；另一方面，当Binder驱动程序主动请求进程注册一个新的线程到它的Binder线程池中来处理进程间通信请求之后，新创建的线程就会使用命令协议代码BC_REGISTER_LOOPER来通知Binder驱动程序，它准备就绪了。最后，当一个线程要退出时，它就使用命令协议代码BC_EXIT_LOOPER从Binder驱动程序中注销，这样它就不会再接收到进程间通信请求了。

命令协议代码BC_REQUEST_DEATH_NOTIFICATION和BC_CLEAR_DEATH_NOTIFICATION后面跟的通信数据使用一个结构体binder_ptr_cookie来描述。一方面，如果一个进程希望获得它所引用的Service组件的死亡接收通知，那么它就需要使用命令协议代码BC_REQUEST_DEATH_NOTIFICATION来向Binder驱动程序注册一个死亡接收通知；另一方面，如果一个进程想注销之前所注册的一个死亡

接收通知，那么它就需要使用命令协议代码BC_CLEAR_DEATH_NOTIFICATION来向Binder驱动程序发出请求。

命令协议代码BC_DEAD_BINDER_DONE后面跟的通信数据是一个void类型的指针，指向一个死亡接收通知结构体binder_ref_death的地址。当一个进程获得一个Service组件的死亡通知时，它就会使用命令协议代码BC_DEAD_BINDER_DONE来通知Binder驱动程序，它已经处理完成该Service组件的死亡通知了。

返回协议代码的定义如下所示。

enum BinderDriverReturnProtocol

kernel/goldfish/drivers/staging/android/binder.h

```
01 enum BinderDriverReturnProtocol {
02         BR_ERROR = _IOR('r', 0, int),
03         /*
04          * int: error code
05          */
06
07         BR_OK = _IO('r', 1),
08         /* No parameters! */
09
10         BR_TRANSACTION = _IOR('r', 2, struct binder_transaction_data),
11         BR_REPLY = _IOR('r', 3, struct binder_transaction_data),
12         /*
13          * binder_transaction_data: the received command.
14          */
15
16         BR_ACQUIRE_RESULT = _IOR('r', 4, int),
17         /*
18          * not currently supported
19          * int: 0 if the last bcATTEMPT_ACQUIRE was not successful.
20          * Else the remote object has acquired a primary reference.
21          */
22
23         BR_DEAD_REPLY = _IO('r', 5),
24         /*
25          * The target of the last transaction (either a bcTRANSACTION or
26          * a bcATTEMPT_ACQUIRE) is no longer with us.  No parameters.
27          */
28
29         BR_TRANSACTION_COMPLETE = _IO('r', 6),
30         /*
31          * No parameters... always refers to the last transaction requested
32          * (including replies).  Note that this will be sent even for
33          * asynchronous transactions.
34          */
35
36         BR_INCREFS = _IOR('r', 7, struct binder_ptr_cookie),
37         BR_ACQUIRE = _IOR('r', 8, struct binder_ptr_cookie),
38
39         BR_RELEASE = _IOR('r', 9, struct binder_ptr_cookie),
40         BR_DECREFS = _IOR('r', 10, struct binder_ptr_cookie),
41         /*
42          * void *:      ptr to binder
43          * void *: cookie for binder
44          */
45
46         BR_ATTEMPT_ACQUIRE = _IOR('r', 11, struct binder_pri_ptr_cookie),
47         /*
48          * not currently supported
49          * int: priority
```

```
50          * void *: ptr to binder
51          * void *: cookie for binder
52          */
53
54         BR_NOOP = _IO('r', 12),
55         /*
56          * No parameters.  Do nothing and examine the next command.  It exists
57          * primarily so that we can replace it with a BR_SPAWN_LOOPER command.
58          */
59
60         BR_SPAWN_LOOPER = _IO('r', 13),
61         /*
62          * No parameters.  The driver has determined that a process has no
63          * threads waiting to service incomming transactions.  When a process
64          * receives this command, it must spawn a new service thread and
65          * register it via bcENTER_LOOPER.
66          */
67
68         BR_FINISHED = _IO('r', 14),
69         /*
70          * not currently supported
71          * stop threadpool thread
72          */
73
74
75         BR_DEAD_BINDER = _IOR('r', 15, void *),
76         /*
77          * void *: cookie
78          */
79         BR_CLEAR_DEATH_NOTIFICATION_DONE = _IOR('r', 16, void *),
80         /*
81          * void *: cookie
82          */
83
84         BR_FAILED_REPLY = _IO('r', 17),
85         /*
86          * The the last transaction (either a bcTRANSACTION or
87          * a bcATTEMPT_ACQUIRE) failed (e.g. out of memory).  No parameters.
88          */
89 };
```

返回协议代码BR_ERROR后面跟的通信数据是一个整数，用来描述一个错误代码。Binder驱动程序在处理应用程序进程发出的某个请求时，如果发生了异常情况，它就会使用返回协议代码BR_ERROR来通知应用程序进程。

返回协议代码BR_OK后面不需要指定通信数据。Binder驱动程序成功处理了应用程序进程发出的某一个请求之后，它就会使用返回协议代码BR_OK来通知应用程序进程。

返回协议代码BR_TRANSACTION和BR_REPLY后面跟的通信数据使用一个结构体binder_transaction_data来描述。当一个Client进程向一个Server进程发出进程间通信请求时，Binder驱动程序就会使用返回协议代码BR_TRANSACTION通知该Server进程来处理该进程间通信请求；当Server进程处理完成该进程间通信请求之后，Binder驱动程序就会使用返回协议代码BR_REPLY将进程间通信请求结果数据返回给Client进程。

返回协议代码BR_ACQUIRE_RESULT在当前的Binder驱动程序实现中不支持。

返回协议代码BR_DEAD_REPLY后面不需要指定通信数据。Binder驱动程序在处理进程间通信请求时，如果发现目标进程或者目标线程已经死亡，它就会使用返回协议代码BR_DEAD_REPLY来通知源进程。

返回协议代码BR_TRANSACTION_COMPLETE后面不需要指定通信数据。当Binder驱动程序接收

到应用程序进程给它发送的一个命令协议代码BC_TRANSACTION或者BC_REPLY时，它就会使用返回协议代码BR_TRANSACTION_COMPLETE来通知应用程序进程，该命令协议代码已经被接收，正在分发给目标进程或者目标线程处理。

返回协议代码BR_INCREFS、BR_ACQUIRE、BR_RELEASE和BR_DECREFS后面跟的通信数据使用一个结构体binder_ptr_cookie来描述，其中，返回协议代码BR_INCREFS和BR_DECREFS分别用来增加和减少一个Service组件的弱引用计数；而返回协议代码BR_ACQUIRE和BR_RELEASE分别用来增加和减少一个Service组件的强引用计数。

返回协议代码BR_ATTEMPT_ACQUIRE在当前的Binder驱动程序实现中不支持。

返回协议代码BR_NOOP后面不需要指定通信数据。Binder驱动程序使用返回协议代码BR_NOOP来通知应用程序进程执行一个空操作，它的存在是为了方便以后可以替换为返回协议代码BR_SPAWN_LOOPER。

返回协议代码BR_SPAWN_LOOPER后面不需要指定通信数据。当Binder驱动程序发现一个进程没有足够的空闲Binder线程来处理进程间通信请求时，它就会使用返回协议代码BR_SPAWN_LOOPER来通知该进程增加一个新的线程到Binder线程池中。

返回协议代码BR_FINISHED在当前的Binder驱动程序实现中不支持。

返回协议代码BR_DEAD_BINDER和BR_CLEAR_DEATH_NOTIFICATION_DONE后面跟的通信数据是一个void类型的指针，它指向一个用来接收Service组件死亡通知的对象的地址。当Binder驱动程序监测到一个Service组件的死亡事件时，它就会使用返回协议代码BR_DEAD_BINDER来通知相应的Client进程。当Client进程通知Binder驱动程序注销它之前所注册的一个死亡接收通知时，Binder驱动程序执行完成这个注销操作之后，就会使用返回协议代码BR_CLEAR_DEATH_NOTIFICATION_DONE来通知Client进程。

返回协议代码BR_FAILED_REPLY后面不需要指定通信数据。当Binder驱动程序处理一个进程发出的BC_TRANSACTION命令协议时，如果发生了异常情况，它就会使用返回协议代码BR_FAILED_REPLY来通知源进程。

介绍完Binder驱动程序提供的命令协议代码和返回协议代码之后，接下来我们继续分析这些协议所使用的两个结构体binder_ptr_cookie和binder_transaction_data的定义。

struct binder_ptr_cookie

kernel/goldfish/drivers/staging/android/binder.h

```
1 struct binder_ptr_cookie {
2         void *ptr;
3         void *cookie;
4 };
```

结构体binder_ptr_cookie用来描述一个Binder实体对象或者一个Service组件的死亡接收通知。当结构体binder_ptr_cookie描述的是一个Binder实体对象时，成员变量ptr和cookie的含义等同于前面所介绍的结构体binder_node的成员变量ptr和cookie；当结构体binder_ptr_cookie描述的是一个Service组件的死亡接收通知时，成员变量ptr指向的是一个Binder引用对象的句柄值，而成员变量cookie指向的是一个用来接收死亡通知的对象的地址。

struct binder_transaction_data

kernel/goldfish/drivers/staging/android/binder.h

```
01 struct binder_transaction_data {
02         /* The first two are only used for bcTRANSACTION and brTRANSACTION,
03          * identifying the target and contents of the transaction.
```

```
04          */
05         union {
06                 size_t  handle; /* target descriptor of command transaction */
07                 void    *ptr;   /* target descriptor of return transaction */
08         } target;
09         void            *cookie;        /* target object cookie */
10         unsigned int    code;           /* transaction command */
11
12         /* General information about the transaction. */
13         unsigned int    flags;
14         pid_t           sender_pid;
15         uid_t           sender_euid;
16         size_t          data_size;      /* number of bytes of data */
17         size_t          offsets_size;   /* number of bytes of offsets */
18
19         /* If this transaction is inline, the data immediately
20          * follows here; otherwise, it ends with a pointer to
21          * the data buffer.
22          */
23         union {
24                 struct {
25                         /* transaction data */
26                         const void      *buffer;
27                         /* offsets from buffer to flat_binder_object structs */
28                         const void      *offsets;
29                 } ptr;
30                 uint8_t buf[8];
31         } data;
32 };
```

结构体binder_transaction_data用来描述进程间通信过程中所传输的数据。

成员变量target是一个联合体，用来描述一个目标Binder实体对象或者目标Binder引用对象。如果它描述的是一个目标Binder实体对象，那么它的成员变量ptr就指向与该Binder实体对象对应的一个Service组件内部的一个弱引用计数对象（weakref_impl）的地址；如果它描述的是一个目标Binder引用对象，那么它的成员变量handle就指向该Binder引用对象的句柄值。

成员变量cookie是由应用程序进程指定的额外参数。当Binder驱动程序使用返回命令协议BR_TRANSACTION向一个Server进程发出一个进程间通信请求时，这个成员变量才有实际意义，它指向的是目标Service组件的地址。

成员变量code是由执行进程间通信的两个进程互相约定好的一个通信代码，Binder驱动程序完全不关心它的含义。

成员变量flags是一个标志值，用来描述进程间通信行为特征，它的取值如下所示。

kernel/goldfish/drivers/staging/android/binder.h

```
1 enum transaction_flags {
2         TF_ONE_WAY         = 0x01, /* this is a one-way call: async, no return */
3         TF_ROOT_OBJECT     = 0x04, /* contents are the component's root object */
4         TF_STATUS_CODE     = 0x08, /* contents are a 32-bit status code */
5         TF_ACCEPT_FDS      = 0x10, /* allow replies with file descriptors */
6 };
```

目前，只使用了TF_ONE_WAY、TF_STATUS_CODE和TF_ACCEPT_FDS这三个标志值。如果成员变量flags的TF_ONE_WAY位被设置为1，就表示这是一个异步的进程间通信过程；如果成员变量flags的TF_ACCEPT_FDS位被设置为0，就表示源进程不允许目标进程返回的结果数据中包含有文件描述符；如果成员变量flags的TF_STATUS_CODE位被设置为1，就表示成员变量data所描述的数据缓冲区的内容是一个4字节的状态码。

成员变量sender_pid和sender_euid表示发起进程间通信请求的进程的PID和UID。这两个成员变量

的值是由Binder驱动程序来填写的，因此，目标进程通过这两个成员变量就可以识别出源进程的身份，以便进行安全检查。

成员变量data_size和offsets_size分别用来描述一个通信数据缓冲区以及一个偏移数组的大小。成员变量data是一个联合体，它指向一个通信数据缓冲区。当通信数据较小时，就直接使用联合体内静态分配的数组buf来传输数据；当通信数据较大时，就需要使用一块动态分配的缓冲区来传输数据了。这块动态分配的缓冲区通过一个包含两个指针的结构体来描述，即通过联合体内的成员变量ptr来描述。结构体ptr的成员变量buffer指向一个数据缓冲区，它是真正用来保存通信数据的，它的大小由前面所描述的成员变量data_size来指定。当数据缓冲区中包含有Binder对象时，那么紧跟在这个数据缓冲区的后面就会有一个偏移数组offsets，用来描述数据缓冲区中每一个Binder对象的位置。有了这个偏移数组之后，Binder驱动程序就可以正确地维护其内部的Binder实体对象和Binder引用对象的引用计数。

数据缓冲区中的每一个Binder对象都使用一个flat_binder_object结构体来描述。下面我们就通过一个例子来描述数据缓冲区的内存布局，如图5-3所示。

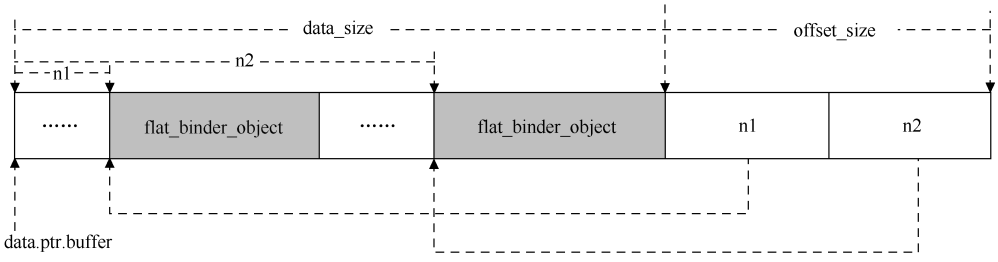


图5-3　Binder进程间通信数据缓冲区的内存布局

在图5-3所描述的数据缓冲区中，有两个Binder对象，相应地，偏移数组的大小就等于2，里面保存的就是这两个Binder对象在数据缓冲区中的位置n1和n2。

下面我们继续分析结构体flat_binder_object的定义。

struct flat_binder_object

kernel/goldfish/drivers/staging/android/binder.h

```
01 struct flat_binder_object {
02         /* 8 bytes for large_flat_header. */
03         unsigned long           type;
04         unsigned long           flags;
05
06         /* 8 bytes of data. */
07         union {
08                 void            *binder;        /* local object */
09                 signed long     handle;         /* remote object */
10         };
11
12         /* extra data associated with local object */
13         void                    *cookie;
14 };
```

结构体flat_binder_object除了可以描述一个Binder实体对象和一个Binder引用对象之外，还可以用来描述一个文件描述符，它们是通过成员变量type来加以区别的。

成员变量type的取值范围如下所示。

kernel/goldfish/drivers/staging/android/binder.h

```
01 #define B_PACK_CHARS(c1, c2, c3, c4) \
02         ((((c1)<<24)) | (((c2)<<16)) | (((c3)<<8)) | (c4))
03 #define B_TYPE_LARGE 0x85
```

```
04
05 enum {
06         BINDER_TYPE_BINDER      = B_PACK_CHARS('s', 'b', '*', B_TYPE_LARGE),
07         BINDER_TYPE_WEAK_BINDER = B_PACK_CHARS('w', 'b', '*', B_TYPE_LARGE),
08         BINDER_TYPE_HANDLE      = B_PACK_CHARS('s', 'h', '*', B_TYPE_LARGE),
09         BINDER_TYPE_WEAK_HANDLE = B_PACK_CHARS('w', 'h', '*', B_TYPE_LARGE),
10         BINDER_TYPE_FD          = B_PACK_CHARS('f', 'd', '*', B_TYPE_LARGE),
11 };
```

其中，BINDER_TYPE_BINDER和BINDER_TYPE_WEAK_BINDER都是用来描述一个Binder实体对象的，前者描述的是一个强类型的Binder实体对象，而后者描述的是一个弱类型的Binder实体对象；BINDER_TYPE_HANDLE和BINDER_TYPE_WEAK_HANDLE用来描述一个Binder引用对象，前者描述的是一个强类型的Binder引用对象，而后者描述的是一个弱类型的Binder引用对象；BINDER_TYPE_FD用来描述一个文件描述符。

成员变量flags是一个标志值，只有当结构体flat_binder_object描述的是一个Binder实体对象时，它才有实际意义。目前只用到该成员变量的第0位到第8位。其中，第0位到第7位描述的是一个Binder实体对象在处理一个进程间通信请求时，它所运行在的线程应当具有的最小线程优先级；第8位用来描述一个Binder实体对象是否可以将一块包含有文件描述符的数据缓冲区传输给目标进程，如果它的值等于1，就表示允许，否则就不允许。

成员变量binder和handle组成了一个联合体。当结构体flat_binder_object描述的是一个Binder实体对象时，那么就使用成员变量binder来指向与该Binder实体对象对应的一个Service组件内部的一个弱引用计数对象（weakref_impl）的地址，并且使用成员变量cookie来指向该Service组件的地址；当结构体flat_binder_object描述的是一个Binder引用对象时，那么就使用成员变量handle来描述该Binder引用对象的句柄值。

至此，Binder驱动程序的基础数据结构就介绍完了。接下来，我们开始分析Binder设备的初始化过程。

5.1.2 Binder设备的初始化过程

Binder设备的初始化过程是在Binder驱动程序的初始化函数binder_init中进行的，它的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static int __init binder_init(void)
02 {
03         int ret;
04
05         binder_proc_dir_entry_root = proc_mkdir("binder", NULL);
06         if (binder_proc_dir_entry_root)
07                 binder_proc_dir_entry_proc = proc_mkdir("proc", binder_proc_dir_entry_root);
08         ret = misc_register(&binder_miscdev);
09         if (binder_proc_dir_entry_root) {
10                 create_proc_read_entry("state", S_IRUGO, binder_proc_dir_entry_root, binder_
read_proc_state, NULL);
11                 create_proc_read_entry("stats", S_IRUGO, binder_proc_dir_entry_root, binder_
read_proc_stats, NULL);
12                  create_proc_read_entry("transactions", S_IRUGO, binder_proc_dir_entry_root,
binder_read_proc_transactions, NULL);
13                   create_proc_read_entry("transaction_log", S_IRUGO, binder_proc_dir_entry_
root, binder_read_proc_transaction_log, &binder_transaction_log);
14                  create_proc_read_entry("failed_transaction_log", S_IRUGO, binder_proc_dir_
entry_root, binder_read_proc_transaction_log, &binder_transaction_log_failed);
15         }
16         return ret;
17 }
```

第5行到第7行代码在目标设备上创建了一个/proc/binder/proc目录，每一个使用了Binder进程间通信机制的进程在该目录下都对应有一个文件，这些文件是以进程ID来命名的，通过它们就可以读取到各个进程的Binder线程池、Binder实体对象、Binder引用对象以及内核缓冲区等信息。接下来第9行到第15行的if语句块又在/proc/binder目录下创建了五个文件state、stats、transactions、transaction_log和failed_transaction_log，通过这五个文件就可以读取到Binder驱动程序的运行状况。例如，各个命令协议（BinderDriverCommandProtocol）和返回协议（BinderDriverReturnProtocol）的请求次数、日志记录信息，以及正在执行进程间通信过程的进程信息等。

第8行调用函数misc_register来创建一个Binder设备。在前面4.2.2小节介绍日志设备的初始化过程时，我们已经分析过函数misc_register的实现了，它是用来创建一个misc类型的字符设备的。这里我们主要分析全局变量binder_miscdev的定义。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static struct file_operations binder_fops = {
02         .owner = THIS_MODULE,
03         .poll = binder_poll,
04         .unlocked_ioctl = binder_ioctl,
05         .mmap = binder_mmap,
06         .open = binder_open,
07         .flush = binder_flush,
08         .release = binder_release,
09 };
10
11 static struct miscdevice binder_miscdev = {
12         .minor = MISC_DYNAMIC_MINOR,
13         .name = "binder",
14         .fops = &binder_fops
15 };
```

Binder驱动程序在目标设备上创建了一个Binder设备文件/dev/binder，这个设备文件的操作方法列表是由全局变量binder_fops指定的。全局变量binder_fops为Binder设备文件/dev/binder指定文件打开、内存映射和IO控制函数分别为binder_open、binder_mmap和binder_ioctl。

在接下来的5.1.3和5.1.4小节中，我们将详细分析函数binder_open和binder_mmap的实现；在5.6到5.9小节中，我们将结合具体的使用情景来分析函数binder_ioctl的实现。

5.1.3　Binder设备文件的打开过程

一个进程在使用Binder进程间通信机制之前，首先要调用函数open打开设备文件/dev/binder来获得一个文件描述符，然后才能通过这个文件描述符来和Binder驱动程序交互，继而和其他进程执行Binder进程间通信。

从前面5.1.2小节的内容可以知道，当进程调用函数open打开设备文件/dev/binder时，Binder驱动程序中的函数binder_open就会被调用，它的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static int binder_open(struct inode *nodp, struct file *filp)
02 {
03         struct binder_proc *proc;
04
05         if (binder_debug_mask & BINDER_DEBUG_OPEN_CLOSE)
06            printk(KERN_INFO "binder_open: %d:%d\n", current->group_leader->pid, current->pid);
07
08         proc = kzalloc(sizeof(*proc), GFP_KERNEL);
09         if (proc == NULL)
```

```
10                  return -ENOMEM;
11          get_task_struct(current);
12          proc->tsk = current;
13          INIT_LIST_HEAD(&proc->todo);
14          init_waitqueue_head(&proc->wait);
15          proc->default_priority = task_nice(current);
16          mutex_lock(&binder_lock);
17          binder_stats.obj_created[BINDER_STAT_PROC]++;
18          hlist_add_head(&proc->proc_node, &binder_procs);
19          proc->pid = current->group_leader->pid;
20          INIT_LIST_HEAD(&proc->delivered_death);
21          filp->private_data = proc;
22          mutex_unlock(&binder_lock);
23
24          if (binder_proc_dir_entry_proc) {
25              char strbuf[11];
26              snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
27              remove_proc_entry(strbuf, binder_proc_dir_entry_proc);
28              create_proc_read_entry(strbuf, S_IRUGO, binder_proc_dir_entry_proc, binder_read_proc_proc, proc);
29          }
30
31          return 0;
32  }
```

第8行首先为进程创建一个binder_proc结构体proc，接下来第11行到第20行代码对该binder_proc结构体proc进行初始化。在初始化binder_proc结构体proc的过程中，第12行、第15行和第19行代码分别使用进程的任务控制块current、进程优先级task_nice(current)和进程组ID来初始化binder_proc结构体proc的成员变量tsk、default_priority和pid。

第18行将binder_proc结构体proc加入到一个全局hash队列binder_procs中。全局hash队列变量binder_procs的定义如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
1 static HLIST_HEAD(binder_procs);
```

Binder驱动程序将所有打开了设备文件/dev/binder的进程都加入到全局hash队列binder_procs中，因此，通过遍历这个hash队列就可以知道系统当前有多少个进程在使用Binder进程间通信机制。

第21行将初始化完成之后的binder_proc结构体proc保存在参数filp的成员变量private_data中。参数filp指向一个打开文件结构体，当进程调用函数open打开设备文件/dev/binder之后，内核就会返回一个文件描述符给进程，而这个文件描述符与参数filp所指向的打开文件结构体是关联在一起的。因此，当进程后面以这个文件描述符为参数调用函数mmap或者ioctl来与Binder驱动程序交互时，内核就会将与该文件描述符相关联的打开文件结构体传递给Binder驱动程序，这时候Binder驱动程序就可以通过它的成员变量private_data来获得前面在函数binder_open中为进程创建的binder_proc结构体proc。

最后，第24行到第29行的if语句块在目标设备上的/proc/binder/proc目录下创建一个以进程ID为名称的只读文件，并且以函数binder_read_proc_proc作为它的文件内容读取函数。通过读取文件/proc/binder/proc/<PID>的内容，我们就可以获得进程<PID>的Binder线程池、Binder实体对象、Binder引用对象，以及内核缓冲区等信息。

5.1.4 Binder设备文件的内存映射过程

进程打开了设备文件/dev/binder之后，还必须要调用函数mmap把这个设备文件映射到进程的地址空间，然后才可以使用Binder进程间通信机制。设备文件/dev/binder对应的是一个虚拟的设备，将它映射到进程的地址空间的目的并不是对它的内容感兴趣，而是为了为进程分配内核缓冲区，以便它可

以用来传输进程间通信数据。

从前面5.1.2小节的内容可以知道，当进程调用函数mmap将设备文件/dev/binder映射到自己的地址空间时，Binder驱动程序中的函数binder_mmap就会被调用。函数binder_mmap比较长，我们分两段来阅读。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static int binder_mmap(struct file *filp, struct vm_area_struct *vma)
02 {
03         int ret;
04         struct vm_struct *area;
05         struct binder_proc *proc = filp->private_data;
06         const char *failure_string;
07         struct binder_buffer *buffer;
08
09         if ((vma->vm_end - vma->vm_start) > SZ_4M)
10                 vma->vm_end = vma->vm_start + SZ_4M;
11
12         ......
13
14         if (vma->vm_flags & FORBIDDEN_MMAP_FLAGS) {
15                 ret = -EPERM;
16                 failure_string = "bad vm_flags";
17                 goto err_bad_arg;
18         }
19         vma->vm_flags = (vma->vm_flags | VM_DONTCOPY) & ~VM_MAYWRITE;
20
21         if (proc->buffer) {
22                 ret = -EBUSY;
23                 failure_string = "already mapped";
24                 goto err_already_mapped;
25         }
```

参数vma指向一个结构体vm_area_struct，用来描述一段虚拟地址空间。此外，第4行定义的变量area指向的是一个结构体vm_struct，这也是用来描述一段虚拟地址空间的。这两个结构体所描述的虚拟地址空间有什么区别呢？我们首先需要明确的是，结构体vm_area_struct和vm_struct所描述的虚拟地址空间是连续的，但是它们对应的物理页面是可以不连续的。在Linux内核中，一个进程可以占用的虚拟地址空间是4G，其中，0~3G是用户地址空间，3G~4G是内核地址空间。为了区分进程的用户地址空间和内核地址空间，Linux内核就分别使用结构体vm_area_struct和vm_struct来描述它们。实际上，结构体vm_struct所描述的内核地址空间范围只有(3G + 896M + 8M)~4G。那么，中间空出来的3G~(3G + 896M + 8M)是用做什么的呢？原来，这部分内核地址空间是有特殊用途的。其中，3G~(3G + 896M)的896M空间是用来映射物理内存的前896M的，它们之间有简单的线性对应关系；而(3G + 896M)~(3G + 896M + 8M)的8M空间是一个安全保护区，是用来检测非法指针的，即所有指向这8M空间的指针都是非法的[1]。

参数filp指向一个打开文件结构体，它的成员变量private_data指向的是一个进程结构体binder_proc，它是在Binder驱动程序的函数binder_open中创建的。因此，第5行可以安全地将参数filp的成员变量private_data转换为一个binder_proc结构体指针，并且保存在变量proc中。参数vma的成员变量vm_start和vm_end指定了要映射的用户地址空间范围，第9行判断它是否超过4M。如果是的话，那么第10行就将它截断为4M。从这里就可以看出，Binder驱动程序最多可以为进程分配4M内核缓冲区来传输进程间通信数据。

接下来第14行检查进程要映射的用户地址空间是否可写，其中，FORBIDDEN_MMAP_FLAGS是

1　Linux进程地址空间的相关知识，可以参考*Understanding the Linux Kernel*一书的第8章——Memory Management。

一个宏，它的定义如下：

kernel/goldfish/drivers/staging/android/binder.c

```
1 #define FORBIDDEN_MMAP_FLAGS                (VM_WRITE)
```

Binder驱动程序为进程分配的内核缓冲区在用户空间只可以读，而不可以写，因此，如果进程指定要映射的用户地址空间可写，第17行就出错返回了。

进程指定要映射的用户地址空间除了不可以写之外，也是不可以复制，以及禁止设置可能会执行写操作标志位的，因此，第19行代码就将参数vma的成员变量vm_flags的VM_DONTCOPY位设置为1，并且将VM_MAYWRITE位设置为0。

第21行的if语句判断进程是否重复调用函数mmap来映射设备文件/dev/binder，即binder_proc结构体proc的成员变量buffer是否已经指向一块内核缓冲区。如果是，第24行就出错返回了。

函数binder_mmap接下来就开始为进程分配内核缓冲区了，如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
26         area = get_vm_area(vma->vm_end - vma->vm_start, VM_IOREMAP);
27         if (area == NULL) {
28                 ret = -ENOMEM;
29                 failure_string = "get_vm_area";
30                 goto err_get_vm_area_failed;
31         }
32         proc->buffer = area->addr;
33         proc->user_buffer_offset = vma->vm_start - (uintptr_t)proc->buffer;
34
35     ......
36
37         proc->pages = kzalloc(sizeof(proc->pages[0]) * ((vma->vm_end - vma->vm_start)/PAGE_SIZE), GFP_KERNEL);
38         if (proc->pages == NULL) {
39                 ret = -ENOMEM;
40                 failure_string = "alloc page array";
41                 goto err_alloc_pages_failed;
42         }
43         proc->buffer_size = vma->vm_end - vma->vm_start;
44
45         vma->vm_ops = &binder_vm_ops;
46         vma->vm_private_data = proc;
47
48         if (binder_update_page_range(proc, 1, proc->buffer, proc->buffer + PAGE_SIZE, vma)) {
49                 ret = -ENOMEM;
50                 failure_string = "alloc small buf";
51                 goto err_alloc_small_buf_failed;
52         }
53         buffer = proc->buffer;
54         INIT_LIST_HEAD(&proc->buffers);
55         list_add(&buffer->entry, &proc->buffers);
56         buffer->free = 1;
57         binder_insert_free_buffer(proc, buffer);
58         proc->free_async_space = proc->buffer_size / 2;
59         barrier();
60         proc->files = get_files_struct(current);
61         proc->vma = vma;
62
63           /*printk(KERN_INFO "binder_mmap: %d %lx-%lx maps %p\n", proc->pid, vma->vm_start,
vma->vm_end, proc->buffer);*/
64         return 0;
65
66         ......
67 }
```

第26行调用函数get_vm_area在进程的内核地址空间中分配一段大小为（vma->vm_end - vma->

vm_start）的空间。如果分配成功，就把它的起始地址以及大小保存在proc->buffer和proc->buffer_size中，并且指定它的打开和关闭函数为binder_vma_open和binder_vma_close。

kernel/goldfish/drivers/staging/android/binder.c

```
1 static struct vm_operations_struct binder_vm_ops = {
2         .open = binder_vma_open,
3         .close = binder_vma_close,
4 };
```

第33行计算要映射的用户空间起始地址与前面获得的内核空间起始地址的差值，并且保存在proc->user_buffer_offset中。Binder驱动程序为进程分配的内核缓冲区有两个地址，其中一个是用户空间地址，由参数vma所指向的一个vm_area_struct结构体来描述；另一个是内核空间地址，由变量area所指向的一个vm_struct结构体来描述。进程通过用户空间地址来访问这块内核缓冲区的内容，而Binder驱动程序通过内核空间地址来访问这块内核缓冲区的内容。由于它们是连续的，并且起始地址相差一个固定值，因此，只要知道其中的一个地址，就可以方便地计算出另外一个地址。

函数接下来就要为进程要映射的虚拟地址空间vma和area分配物理页面了，即分配内核缓冲区。第37行首先创建了一个物理页面结构体指针数组，大小为(vma->vm_end - vma->vm_start)/PAGE_SIZE，即每一页虚拟地址空间都对应有一个物理页面，并且将该数组的地址保存在proc->pages中。在Linux内核中，一个物理页面的大小为PAGE_SIZE。PAGE_SIZE是一个宏，一般定义为4K。第48行接着调用函数binder_update_page_range来为虚拟地址空间area分配一个物理页面，对应的内核地址空间为proc->buffer ~ (proc->buffer + PAGE_SIZE)。这个物理页面分配成功之后，第53行到第55行代码就使用一个binder_buffer结构体buffer来描述它，并且将它加入到进程结构体 proc的内核缓冲区列表buffers中。由于这个物理页面是空闲的，因此，第57行继续调用函数binder_insert_free_buffer将它加入到进程结构体 proc的空闲内核缓冲区红黑树free_buffers中。最后，第58行将进程最大可用于异步事务的内核缓冲区大小设置为总的内核缓冲区大小的一半，这样就可以防止异步事务消耗过多的内核缓冲区，从而影响同步事务的执行。

下面我们继续分析函数binder_update_page_range和binder_insert_free_buffer的实现。

函数binder_update_page_range的作用是为一段指定的虚拟地址空间分配或者释放物理页面，它的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static int binder_update_page_range(struct binder_proc *proc, int allocate,
02         void *start, void *end, struct vm_area_struct *vma)
03 {
04         void *page_addr;
05         unsigned long user_page_addr;
06         struct vm_struct tmp_area;
07         struct page **page;
08         struct mm_struct *mm;
09
10         ......
11
12         if (vma)
13                 mm = NULL;
14         else
15                 mm = get_task_mm(proc->tsk);
16
17         if (mm) {
18                 down_write(&mm->mmap_sem);
19                 vma = proc->vma;
20         }
21
22         if (allocate == 0)
```

```
23                 goto free_range;
24
25         ......
26
27         for (page_addr = start; page_addr < end; page_addr += PAGE_SIZE) {
28                 int ret;
29                 struct page **page_array_ptr;
30                 page = &proc->pages[(page_addr - proc->buffer) / PAGE_SIZE];
31
32                 BUG_ON(*page);
33                 *page = alloc_page(GFP_KERNEL | __GFP_ZERO);
34                 ......
35
36                 tmp_area.addr = page_addr;
37                 tmp_area.size = PAGE_SIZE + PAGE_SIZE /* guard page? */;
38                 page_array_ptr = page;
39                 ret = map_vm_area(&tmp_area, PAGE_KERNEL, &page_array_ptr);
40                 ......
41
42                 user_page_addr =
43                         (uintptr_t)page_addr + proc->user_buffer_offset;
44                 ret = vm_insert_page(vma, user_page_addr, page[0]);
45                 ......
46         }
47         if (mm) {
48                 up_write(&mm->mmap_sem);
49                 mmput(mm);
50         }
51
52         return 0;
53
54 free_range:
55         for (page_addr = end - PAGE_SIZE; page_addr >= start;
56              page_addr -= PAGE_SIZE) {
57                 page = &proc->pages[(page_addr - proc->buffer) / PAGE_SIZE];
58                 if (vma)
59                         zap_page_range(vma, (uintptr_t)page_addr +
60                                 proc->user_buffer_offset, PAGE_SIZE, NULL);
61 err_vm_insert_page_failed:
62                 unmap_kernel_range((unsigned long)page_addr, PAGE_SIZE);
63 err_map_kernel_failed:
64                 __free_page(*page);
65                 *page = NULL;
66 err_alloc_page_failed:
67                 ;
68         }
69 err_no_vma:
70         if (mm) {
71                 up_write(&mm->mmap_sem);
72                 mmput(mm);
73         }
74         return -ENOMEM;
75 }
```

第一个参数proc指向要操作的目标进程；第二个参数allocate的值如果等于0，就表示要释放物理页面，否则就表示要分配物理页面；第三参数start和第四个参数end指定了要操作的内核地址空间的开始地址和结束地址；第五个参数vma指向要映射的用户地址空间。第12行到第20行代码判断参数vma是否指向一个空的用户地址空间。如果是，就从目标进程proc的成员变量vma来获得要映射的用户地址空间。

第22行的if语句判断是要为内核地址空间start ~ end分配物理页面还是释放物理页面。如果是分配

物理页面，就执行第27行到第46行的for循环；否则，就执行第55行到第68行的for循环。

我们首先分析物理内存的分配过程。由于内核地址空间start ~ end可能包含了多个页面，因此，第27行到第46行通过一个for循环来依次为每一个虚拟地址空间页面分配一个物理页面。第30行首先从目标进程proc的物理页面结构体指针数组pages中获得一个与内核地址空间page_addr ~ (page_addr + PAGE_SIZE)对应的物理页面指针，接着第33行就调用函数alloc_page为该内核地址空间分配一个物理页面。物理页面分配成功之后，接下来就要分别映射到对应的内核地址空间和用户地址空间。其中，第36行到第39行代码用来映射内核地址空间，而第42行到第44行代码用来映射用户地址空间。

注意

在映射内核地址空间时，第37行将要映射的内核地址空间大小设置为两个页面，即用一个物理页面来映射两个内核地址空间页面。为什么要这样做呢？原来，Linux内核规定，在(3G + 896M + 8M) ~ 4G范围内的任意一块内核地址空间都必须要在后面保留一块空的地址空间来作为安全保护区，用来检测非法指针。因此，第37行就在内核地址空间page_addr ~ (page_addr + PAGE_SIZE)的后面保留了大小为一页的安全保护区。另外一个需要注意的地方是，第43行将内核地址page_addr加上一个偏移值之后，就可以得到与它对应的用户空间地址，这个偏移值保存在目标进程结构体proc的成员变量user_buffer_offset中，它是在前面介绍的函数binder_mmap中设置的。

接着再分析物理页面的释放过程。同样是由于内核地址空间start ~ end可能包含了多个页面，因此，第55行到第68行通过一个for循环来依次为每一个虚拟地址空间页面释放物理页面。第57行首先从目标进程proc的物理页面结构体指针数组pages中获得一个与内核地址空间page_addr ~ (page_addr + PAGE_SIZE)对应的物理页面指针，接着第59行和第62行分别调用函数zap_page_range和unmap_kernel_range来解除该物理页面在用户地址空间和内核地址空间的映射，最后第64行就调用函数__free_page来释放该物理页面。

至此，函数binder_update_page_range的实现就分析完成了。接下来，我们继续分析函数binder_insert_free_buffer的实现，它的作用是将一个空闲的内核缓冲区加入到进程的空闲内核缓冲区红黑树中，如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static void binder_insert_free_buffer(
02         struct binder_proc *proc, struct binder_buffer *new_buffer)
03 {
04         struct rb_node **p = &proc->free_buffers.rb_node;
05         struct rb_node *parent = NULL;
06         struct binder_buffer *buffer;
07         size_t buffer_size;
08         size_t new_buffer_size;
09
10         BUG_ON(!new_buffer->free);
11
12         new_buffer_size = binder_buffer_size(proc, new_buffer);
13
14         ......
15
16         while (*p) {
17                 parent = *p;
18                 buffer = rb_entry(parent, struct binder_buffer, rb_node);
19                 BUG_ON(!buffer->free);
20
21                 buffer_size = binder_buffer_size(proc, buffer);
22
23                 if (new_buffer_size < buffer_size)
```

```
24                          p = &parent->rb_left;
25                  else
26                          p = &parent->rb_right;
27          }
28          rb_link_node(&new_buffer->rb_node, parent, p);
29          rb_insert_color(&new_buffer->rb_node, &proc->free_buffers);
30  }
```

在前面的5.1.1小节中介绍结构体binder_proc时提到，它的成员变量free_buffers用来描述一个红黑树，它按照大小来组织进程中的空闲内核缓冲区。因此，在将内核缓冲区new_buffer加入到目标进程proc的空闲内核缓冲区红黑树中之前，第12行首先调用函数binder_buffer_size来计算它的大小。

函数binder_buffer_size的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
1 static size_t binder_buffer_size(
2          struct binder_proc *proc, struct binder_buffer *buffer)
3 {
4          if (list_is_last(&buffer->entry, &proc->buffers))
5                  return proc->buffer + proc->buffer_size - (void *)buffer->data;
6          else
7                  return (size_t)list_entry(buffer->entry.next,
8                          struct binder_buffer, entry) - (size_t)buffer->data;
9 }
```

使用结构体binder_buffer描述的内核缓冲区由两块数据组成，其中一个是元数据块，用来描述内核缓冲区本身；另一个是有效数据块，用来保存真正的事务数据。一个内核缓冲区的大小指的是其有效数据块的大小。

计算一个内核缓冲区binder_buffer的大小时，需要考虑它在进程内核缓冲区列表buffers中的位置。如果它是列表中的最后一个元素，即第4行的if语句为true，那么它所描述的内核缓冲区的有效数据块就从它的成员变量data开始，一直到Binder驱动程序为进程所分配的一块连续内核地址空间的末尾。因此，第5行就首先计算Binder驱动程序为进程proc所分配的内核地址空间的末尾地址，然后再减去内核缓冲区buffer的成员变量data的地址，最后就可以得到内核缓冲区buffer的有效数据块的大小了，如图5-4所示。

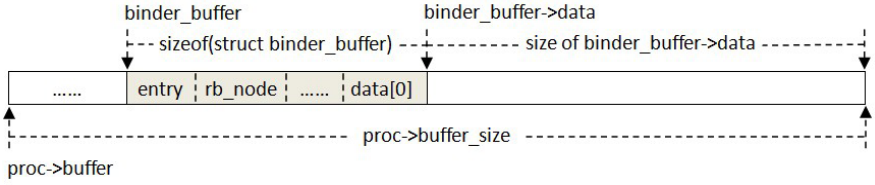


图5-4　位于末尾的内核缓冲区的内存布局

如果内核缓冲区buffer不是进程内核缓冲区列表的最后一个元素，即函数第4行的if语句为false，那么它的大小等于下一个内核缓冲区的起始地址，再减去它的成员变量data的地址，如图5-5所示。

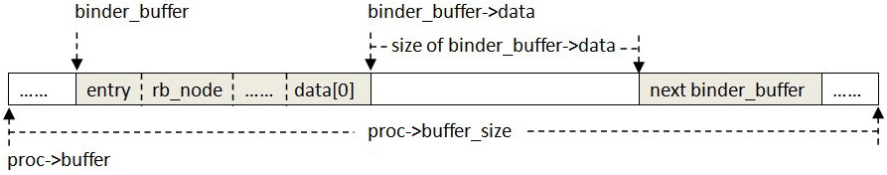


图5-5　位于开头或者中间的内核缓冲区的内存布局

回到函数binder_insert_free_buffer中，得到内核缓冲区new_buffer的大小之后，接下来第16行到第27行的while循环就在目标进程proc的空闲内核缓冲区红黑树free_buffers中找到一个合适位置，最后第

28行和第29行就调用函数rb_link_node和rb_insert_color将内核缓冲区new_buffer保存在这个位置上，这样就相当于将内核缓冲区new_buffer插入到目标进程proc的空闲内核缓冲区红黑树free_buffers中了。

至此，Binder设备文件的内存映射过程就分析完成了。一个使用Binder进程间通信机制的进程只有将Binder设备文件映射到自己的地址空间，Binder驱动程序才能够为它分配内核缓冲区，以便可以用来传输进程间通信数据。Binder驱动程序为进程分配的内核缓冲区有两个地址，其中一个是用户空间地址，另外一个是内核空间地址，它们有简单的线性对应关系，如图5-6所示。

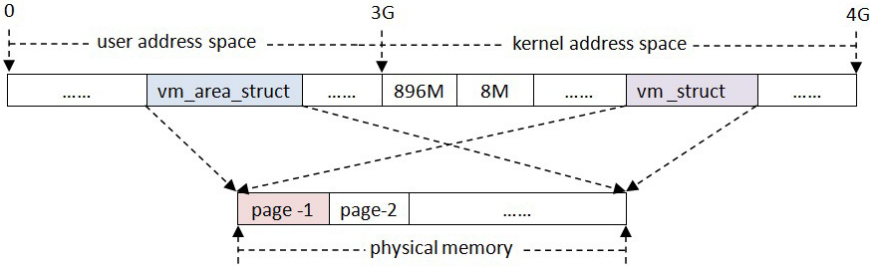


图5-6　Binder设备文件内存映射示意图

Binder驱动程序为进程分配的内核缓冲区即为一系列物理页面，它们分别被映射到进程的用户地址空间和内核地址空间。当Binder驱动程序需要将一块数据传输给一个进程时，它就可以先把这块数据保存在为该进程所分配的一块内核缓冲区中，然后再把这块内核缓冲区的用户空间地址告诉进程，最后进程就可以访问到里面的数据了。这样做的好处便是不需要将数据从内核空间复制到用户空间，从而提高了数据的传输效率。

5.1.5　内核缓冲区管理

开始的时候，Binder驱动程序只为进程分配了一个页面的物理内存，后面会随着进程的需要而分配更多的物理内存，但是最多可以分配4M内存，这是一种按需分配的策略。物理内存的分配是以页面为单位的，但是进程一次使用的内存却不是以页面为单位的，因此，Binder驱动程序为进程维护了一个内核缓冲区池，内核缓冲区池中的每一块内存都使用一个binder_buffer结构体来描述，并且保存在一个列表中。同时，Binder驱动程序又将正在使用的内存块，即已经分配了物理页面的内存块，以及空闲的内存块，即还没有分配物理页面的内存块，分别保存在两个红黑树中。当正在使用的内存块使用完之后，Binder驱动程序就会释放它的物理页面，并且把它加入到空闲内核缓冲区红黑树中；而当进程需要新的内存块时，Binder驱动程序就从空闲内核缓冲区红黑树中分配一块合适的内核缓冲区，并且为它分配物理页面，最后交给进程来使用。

在本节接下来的内容中，我们分别介绍Binder驱动程序是如何管理进程的内核缓冲区的，包括内存缓冲区的分配、释放和查询。

5.1.5.1　分配内核缓冲区

当一个进程使用命令协议BC_TRANSACTION或者BC_REPLY向另外一个进程传递数据时，Binder驱动程序就需要将这些数据从用户空间复制到内核空间，然后再传递给目标进程。这时候Binder驱动程序就需要在目标进程的内存池中分配出一小块内核缓冲区来保存这些数据，以便可以传递给它使用。

在Binder驱动程序中，内核缓冲区的分配操作是由函数binder_alloc_buf实现的，我们分段来阅读。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static struct binder_buffer *binder_alloc_buf(struct binder_proc *proc,
02         size_t data_size, size_t offsets_size, int is_async)
```

```
03 {
04         struct rb_node *n = proc->free_buffers.rb_node;
05         struct binder_buffer *buffer;
06         size_t buffer_size;
07         struct rb_node *best_fit = NULL;
08         void *has_page_addr;
09         void *end_page_addr;
10         size_t size;
11
12         ......
13
14         size = ALIGN(data_size, sizeof(void *)) +
15                 ALIGN(offsets_size, sizeof(void *));
16
17         if (size < data_size || size < offsets_size) {
18                 binder_user_error("binder: %d: got transaction with invalid "
19                         "size %zd-%zd\n", proc->pid, data_size, offsets_size);
20                 return NULL;
21         }
22
23         if (is_async &&
24             proc->free_async_space < size + sizeof(struct binder_buffer)) {
25                 if (binder_debug_mask & BINDER_DEBUG_BUFFER_ALLOC)
26                         printk(KERN_ERR "binder: %d: binder_alloc_buf size %zd f"
27                                 "ailed, no async space left\n", proc->pid, size);
28                 return NULL;
29         }
```

在前面的5.1.1小节中介绍命令协议BinderDriverCommandProtocol时提到，当一个进程使用命令协议BC_TRANSACTION或者BC_REPLY来与Binder驱动程序交互时，它会从用户空间传递一个binder_transaction_data结构体给Binder驱动程序，而在binder_transaction_data结构体中，有一个数据缓冲区和一个偏移数组缓冲区，这两个缓冲区的内容就是需要复制到目标进程的内核缓冲区中的。

了解了这个背景知识之后，我们就可以进一步说明函数的含义了。第一个参数proc用来描述目标进程；第二个参数data_size用来描述数据缓冲区的大小；第三个参数offsets_size用来描述偏移数组缓冲区的大小；第四个参数is_async用来描述所请求的内核缓冲区是用于同步事务还是异步事务的。

第14行和第15行分别将参数data_size和offsets_size对齐到一个void指针大小边界，然后将它们相加就得到要分配的内核缓冲区的大小，并且保存在变量size中。第17行的if语句检查相加后得到的size值是否发生溢出。如果是，就说明请求分配的内核缓冲区太大了，因此，第20行就直接出错返回了。

第23行和第24行的if语句检查要分配的内核缓冲区是否是用于异步事务的。如果是，就要进一步检查请求分配的内核缓冲区的大小是否大于目标进程剩余的可用于异步事务的内核缓冲区的大小。如果是，第28行就直接出错返回了。

注意

当Binder驱动程序为进程分配一个大小为size的内核缓冲区来保存数据时，还要额外分配一个binder_buffer结构体来描述这个内核缓冲区，因此，第24行在判断请求分配的内核缓冲区的大小是否大于目标进程剩余的可用于异步事务的内核缓冲区的大小时，要将变量size的值加上一个binder_buffer结构体的大小。

通过了前面合法性检查之后，我们继续往下分析函数binder_alloc_buf的实现。

kernel/goldfish/drivers/staging/android/binder.c

```
30         while (n) {
31                 buffer = rb_entry(n, struct binder_buffer, rb_node);
32                 BUG_ON(!buffer->free);
```

```
33                 buffer_size = binder_buffer_size(proc, buffer);
34
35                 if (size < buffer_size) {
36                         best_fit = n;
37                         n = n->rb_left;
38                 } else if (size > buffer_size)
39                         n = n->rb_right;
40                 else {
41                         best_fit = n;
42                         break;
43                 }
44         }
45         if (best_fit == NULL) {
46                 printk(KERN_ERR "binder: %d: binder_alloc_buf size %zd failed, "
47                        "no address space\n", proc->pid, size);
48                 return NULL;
49         }
50         if (n == NULL) {
51                 buffer = rb_entry(best_fit, struct binder_buffer, rb_node);
52                 buffer_size = binder_buffer_size(proc, buffer);
53         }
```

第30行到第44行的while循环使用最佳适配算法在目标进程的空闲内核缓冲区红黑树中检查有没有最合适的内核缓冲区可用。如果有，就将它保存在变量best_fit中；否则，第47行就直接出错返回了。如果变量n的值等于NULL，即第50行的if语句为true，那么就说明没能从目标进程的空闲内核缓冲区红黑树中找到一块大小刚刚合适的内核缓冲区，但是找到了一块较大的内核缓冲区，因此，第51行和第52行代码就计算实际找到的内核缓冲区buffer的大小，并且保存在变量buffer_size中。

找到了一块合适的空闲内核缓冲区之后，我们继续往下分析函数binder_alloc_buf的实现。

kernel/goldfish/drivers/staging/android/binder.c

```
54         has_page_addr =
55                 (void *)(((uintptr_t)buffer->data + buffer_size) & PAGE_MASK);
56         if (n == NULL) {
57                 if (size + sizeof(struct binder_buffer) + 4 >= buffer_size)
58                         buffer_size = size; /* no room for other buffers */
59                 else
60                         buffer_size = size + sizeof(struct binder_buffer);
61         }
62         end_page_addr =
63                 (void *)PAGE_ALIGN((uintptr_t)buffer->data + buffer_size);
64         if (end_page_addr > has_page_addr)
65                 end_page_addr = has_page_addr;
66         if (binder_update_page_range(proc, 1,
67             (void *)PAGE_ALIGN((uintptr_t)buffer->data), end_page_addr, NULL))
68                 return NULL;
```

函数第54行和第55行使用宏PAGE_MASK来计算空闲内核缓冲区buffer的结束地址所在的页面的起始地址，并且保存在变量has_page_addr中。其中，宏PAGE_MASK的定义如下所示。

kernel/goldfish/arch/arm/include/asm/page.h

```
1 /* PAGE_SHIFT determines the page size */
2 #define PAGE_SHIFT              12
3 #define PAGE_SIZE               (1UL << PAGE_SHIFT)
4 #define PAGE_MASK               (~(PAGE_SIZE-1))
```

前面提到，如果变量n的值等于NULL，那么就说明空闲内核缓冲区buffer大于要求分配的内核缓冲区。因此，函数就需要对它进行裁剪，裁剪后就得到两块小的内核缓冲区，其中一块用来分配，另外一块要继续留在目示进程的空闲内核缓冲区红黑树中。如果裁剪后得到的第二块内核缓冲区小于或者等于4个字节，那么函数第58行就不对空闲内核缓冲区buffer进行裁剪了，而是把它都分配给目标进

程来使用；如果裁剪后得到的第二块内核缓冲区大于4个字节，那么就需要将它加入到目示进程的空闲内核缓冲区红黑树中。最终将需要分配的内核缓冲区大小保存在变量buffer_size中。

得到了最终要分配的内核缓冲区的大小之后，第62行和第63行就使用宏PAGE_ALIGN将它的结束地址对齐到页面边界，并且保存在变量end_page_addr中。其中，宏PAGE_ALIGN的定义如下所示。

kernel/goldfish/include/linux/mm.h

```
1 /* to align the pointer to the (next) page boundary */
2 #define PAGE_ALIGN(addr) ALIGN(addr, PAGE_SIZE)
```

由于接下来要调用函数binder_update_page_range为前面得到的内核缓冲区分配物理页面，因此，第62行和第63行需要将要分配的内核缓冲区的结束地址对齐到页面边界。对齐之后，得到的地址end_page_addr就有可能大于原来的空闲内核缓冲区buffer的结束地址所在的页面的起始地址has_page_addr，这时候就需要将end_page_addr的值修正为has_page_addr的值，如第64行和第65行代码所示。下面我们通过三种情况来进一步说明地址end_page_addr和has_page_addr的关系。

第一种情况是空闲内核缓冲区buffer的结束地址刚好对齐到页面边界，如图5-7所示。

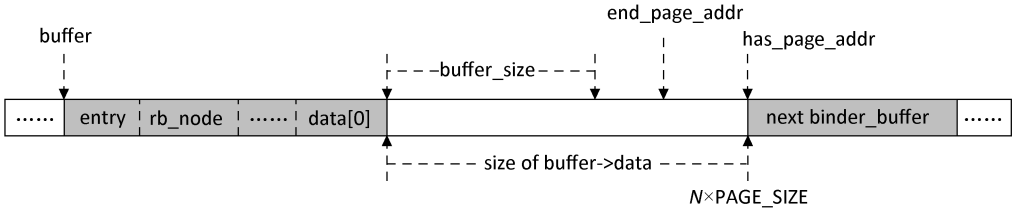


图5-7　内核缓冲区buffer的结束地址对齐到页面边界

这时候要分配的内核缓冲区的结束地址end_page_addr就一定等于或者小于地址has_page_addr，因此，第64行的if语句就为false。

第二种情况是空闲内核缓冲区buffer的结束地址没有对齐到页面边界，这时候要分配的内核缓冲区的结束地址end_page_addr就有可能小于或者大于地址has_page_addr，如图5-8和图5-9所示。

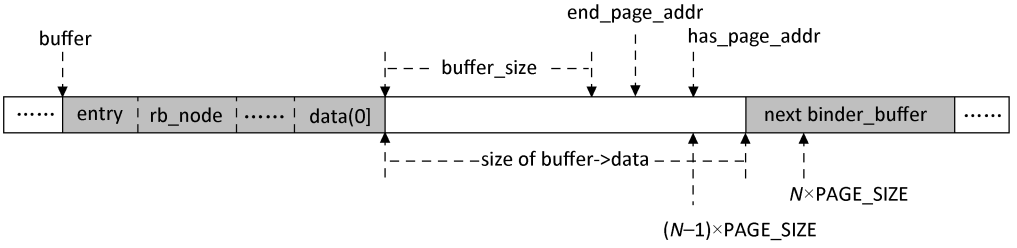


图5-8　内核缓冲区buffer的结束地址没有对齐到页面边界，并且大于要分配的内核缓冲区的结束地址

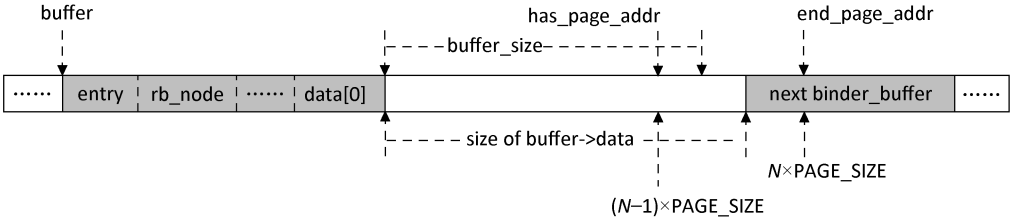


图5-9　内核缓冲区buffer的结束地址没有对齐到页面边界，并且小于要分配的内核缓冲区的结束地址

在图5-8中，虽然空闲内核缓冲区buffer的结束地址没有对齐到页面边界，但是由于它比要分配的内核缓冲区的结束地址大，因此，第64行的if语句也为false。

在图5-9中，当要分配的内核缓冲区大小与空闲内核缓冲区buffer大小相当时，即它们的结束地址都

是在同一个页面时，就会出现地址end_page_addr大于地址has_page_addr的情况。这时候，第64行的if语句就会为true，函数就需要将要分配的内核缓冲区的结束地址end_page_addr的值设置为has_page_addr，因为要分配的内核缓冲区的结束地址end_page_addr所在的页面上肯定分配了物理页面；否则，空闲内核缓冲区buffer后面的一个内核缓冲区就会位于一个无效的物理内存中。另外，假设空闲内核缓冲区buffer是目标进程的最后一个内核缓冲区，那么我们也可以推断出一定不会出现图5-9所示的情况，因为这时候空闲内核缓冲区buffer的结束地址一定是对齐到页面边界的。

计算好要分配的内核缓冲区的结束地址所在的页面地址之后，第66行和第67行就调用函数binder_update_page_range来为它分配物理页面了。函数binder_update_page_range的实现可以参考前面5.1.4小节的内容。

为要分配的内核缓冲区分配好物理页面之后，我们继续往下分析函数binder_alloc_buf的实现。

kernel/goldfish/drivers/staging/android/binder.c

```
69          rb_erase(best_fit, &proc->free_buffers);
70          buffer->free = 0;
71          binder_insert_allocated_buffer(proc, buffer);
72          if (buffer_size != size) {
73                  struct binder_buffer *new_buffer = (void *)buffer->data + size;
74                  list_add(&new_buffer->entry, &buffer->entry);
75                  new_buffer->free = 1;
76                  binder_insert_free_buffer(proc, new_buffer);
77          }
```

第69行首先将空闲内核缓冲区从目标进程的空闲内核缓冲区红黑树中删除，接着第71行调用函数binder_insert_allocated_buffer将前面分配的内核缓冲区加入到目标进程的已分配物理页面的内核缓冲区红黑树中。

函数binder_insert_allocated_buffer的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static void binder_insert_allocated_buffer(
02          struct binder_proc *proc, struct binder_buffer *new_buffer)
03 {
04          struct rb_node **p = &proc->allocated_buffers.rb_node;
05          struct rb_node *parent = NULL;
06          struct binder_buffer *buffer;
07
08          BUG_ON(new_buffer->free);
09
10          while (*p) {
11                  parent = *p;
12                  buffer = rb_entry(parent, struct binder_buffer, rb_node);
13                  BUG_ON(buffer->free);
14
15                  if (new_buffer < buffer)
16                          p = &parent->rb_left;
17                  else if (new_buffer > buffer)
18                          p = &parent->rb_right;
19                  else
20                          BUG();
21          }
22          rb_link_node(&new_buffer->rb_node, parent, p);
23          rb_insert_color(&new_buffer->rb_node, &proc->allocated_buffers);
24 }
```

在前面的5.1.1小节中介绍结构体binder_proc时提到，一个进程的已分配物理页面的内核缓冲区以它们的内核空间地址值作为关键字保存在一个红黑树allocated_buffers中，因此，第10行到第21行的while循环就首先在里面找到一个合适的位置p，接着第22行和第23行就将已经分配了物理页面的内核

缓冲区new_buffer添加到目标进程的红黑树allocated_buffers中。

回到函数binder_alloc_buf中，第72行检查从原来的空闲内核缓冲区中分配出来一块新的内核缓冲区之后，是否还有剩余。如果有，第73行到第76行代码就需要将剩余的内核缓冲区封装成另外一个新的空闲内核缓冲区new_buffer，并且将它添加到目标进程的内核缓冲区列表，以及空闲内核缓冲区红黑树中。

函数binder_alloc_buf将新分配的内核缓冲区返回给调用者之前，还需要对它进行一些初始化操作，如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
78          buffer->data_size = data_size;
79          buffer->offsets_size = offsets_size;
80          buffer->async_transaction = is_async;
81          if (is_async) {
82                  proc->free_async_space -= size + sizeof(struct binder_buffer);
83                  ......
84          }
85
86          return buffer;
87  }
```

第78行和第79行分别设置新分配的内核缓冲区的数据缓冲区和偏移数组缓冲区的大小。第80行设置这个新分配的内核缓冲区是否用于异步事务。如果是，即第81行的语句为true，那么第82行就相应地减少目标进程proc可用于异步事务的内核缓冲区的大小。最后，第86行将新分配的内核缓冲区返回给调用者。

至此，一个新的内核缓冲区的分配过程就分析完成了。接下来，我们继续分析内核缓冲区的释放过程。

5.1.5.2 释放内核缓冲区

当一个进程处理完成Binder驱动程序给它发送的返回协议BR_TRANSACTION或者BR_REPLY之后，它就会使用命令协议BC_FREE_BUFFER来通知Binder驱动程序释放相应的内核缓冲区，以免浪费系统内存。

在Binder驱动程序中，释放内核缓冲区的操作是由函数binder_free_buf实现的，它的定义如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static void binder_free_buf(
02          struct binder_proc *proc, struct binder_buffer *buffer)
03 {
04          size_t size, buffer_size;
05
06          buffer_size = binder_buffer_size(proc, buffer);
07
08          size = ALIGN(buffer->data_size, sizeof(void *)) +
09                  ALIGN(buffer->offsets_size, sizeof(void *));
10          ......
11
12          if (buffer->async_transaction) {
13                  proc->free_async_space += size + sizeof(struct binder_buffer);
14                  ......
15          }
16
17          binder_update_page_range(proc, 0,
18                  (void *)PAGE_ALIGN((uintptr_t)buffer->data),
19                  (void *)(((uintptr_t)buffer->data + buffer_size) & PAGE_MASK),
20                  NULL);
21          rb_erase(&buffer->rb_node, &proc->allocated_buffers);
22          buffer->free = 1;
```

```
23          if (!list_is_last(&buffer->entry, &proc->buffers)) {
24                  struct binder_buffer *next = list_entry(buffer->entry.next,
25                                                  struct binder_buffer, entry);
26                  if (next->free) {
27                          rb_erase(&next->rb_node, &proc->free_buffers);
28                          binder_delete_free_buffer(proc, next);
29                  }
30          }
31          if (proc->buffers.next != &buffer->entry) {
32                  struct binder_buffer *prev = list_entry(buffer->entry.prev,
33                                                  struct binder_buffer, entry);
34                  if (prev->free) {
35                          binder_delete_free_buffer(proc, buffer);
36                          rb_erase(&prev->rb_node, &proc->free_buffers);
37                          buffer = prev;
38                  }
39          }
40          binder_insert_free_buffer(proc, buffer);
41  }
```

第6行计算要释放的内核缓冲区buffer的大小，并且保存在变量buffer_size中；第8行和第9行计算它的数据缓冲区，以及偏移数组缓冲区的大小之后，保存在变量size中。

第12行的if语句检查要释放的内核缓冲区buffer是否是用于异步事务的。如果是，第13行就将它所占用的大小增加到目标进程proc可用于异步事务的内核缓冲区大小free_async_space中。

第17行到第20行调用函数binder_update_page_range释放内核缓冲区buffer用来保存数据的那一部分地址空间所占用的物理页面，接着第21行将它从目标进程proc的已分配物理页面的内核缓冲区红黑树中删除。

如果要释放的内核缓冲区buffer不是目标进程proc的内核缓冲区列表中的最后一个元素，并且它前后的内核缓冲区也是空闲的，那么就需要将它们合并成一个更大的空闲内核缓冲区。第23行到第30行代码和第31行到第39行代码就是用来合并内核缓冲区buffer，以及它后面和前面的空闲内核缓冲区的。合并后得到的内核缓冲区就保存在变量buffer中，因此，第40行就调用函数binder_insert_free_buffer将它添加到目标进程proc的空闲内核缓冲区红黑树中。

合并两个连续的空闲内核缓冲区的实质是将后面一个空闲内核缓冲区删掉，然后将它所占用的地址空间追加到前面一个空闲内核缓冲区中，如图5-10所示。

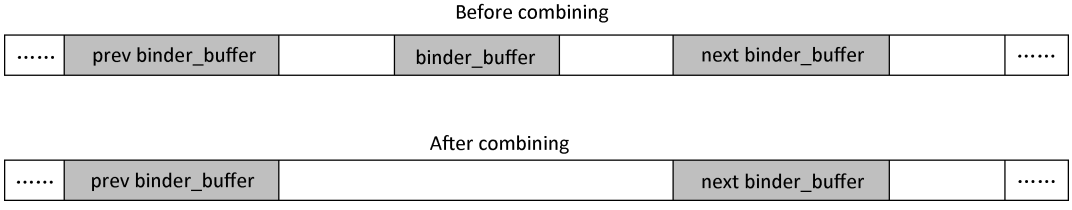


图5-10　合并两个连续的空闲内核缓冲区

注意

从进程的内核缓冲区列表中删除一个内核缓冲区时，需要考虑释放用来描述将要被删除的内核缓冲区的binder_buffer结构体所占用的物理页面。将一个空闲内核缓冲区buffer从进程的内核缓冲区列表中删除是比较简单的，只要调用函数list_del(&buffer->entry)就可以了，但是释放用来描述它的binder_buffer结构体所占用的物理页面就比较复杂了，因为这个binder_buffer结构体可能横跨在两个虚拟地址页面上，如图5-11所示。

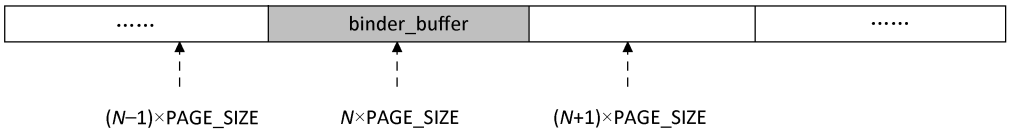


图5–11　横跨在第*N*个和第*N*+1个虚拟地址页面上的binder_buffer结构体

由于Binder驱动程序是以页面大小为单位来分配物理页面的，因此，我们在删除一个空闲缓冲区时，首先需要找到用来描述它的结构体binder_buffer所在的虚拟地址页面的地址。

Binder驱动程序提供了函数buffer_start_page和buffer_end_page来计算一个binder_buffer结构体所占用的虚拟地址页面的地址，如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
1 static void *buffer_start_page(struct binder_buffer *buffer)
2 {
3         return (void *)((uintptr_t)buffer & PAGE_MASK);
4 }
5
6 static void *buffer_end_page(struct binder_buffer *buffer)
7 {
8         return (void *)(((uintptr_t)(buffer + 1) - 1) & PAGE_MASK);
9 }
```

当一个binder_buffer结构体横跨两个虚拟地址页面时，函数buffer_start_page用来计算第一个虚拟地址页面的地址，而函数buffer_end_page用来计算第二个虚拟地址页面的地址；否则，函数buffer_start_page和buffer_end_page返回的是同一个虚拟地址页面的地址。接下来，我们只考虑一个binder_buffer结构体横跨两个虚拟地址页面的情况。

函数buffer_start_page的实现比较容易理解，把一个结构体binder_buffer的地址值与宏PAGE_MASK执行按位与操作，就得到第一个虚拟地址页面的地址了。

函数buffer_end_page的实现相对就复杂一些。由于参数buffer是一个binder_buffer结构体指针，因此，将它的值加1之后，就相当于将它往前移了一个binder_buffer结构体的大小，即得到这个binder_buffer结构体的末尾地址，最后这个末尾地址与宏PAGE_MASK执行按位与操作，就得到第二个虚拟地址页面的地址了。

以图5–11中的binder_buffer结构体为例，我们调用函数buffer_start_page和buffer_end_page来计算它横跨的两个虚拟地址页面的地址，结果如图5–12所示。

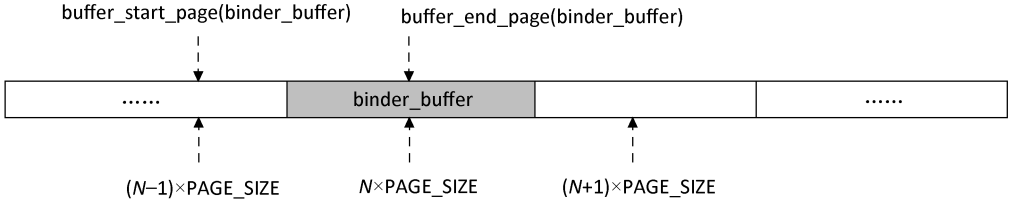


图5–12　计算binder_buffer结构体横跨的两个虚拟地址页面的地址

了解了这些基础知识之后，接下来我们开始分析用来删除binder_buffer结构体的函数binder_delete_free_buffer的实现，如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static void binder_delete_free_buffer(
02         struct binder_proc *proc, struct binder_buffer *buffer)
03 {
04         struct binder_buffer *prev, *next = NULL;
05         int free_page_end = 1;
```

```
06          int free_page_start = 1;
07
08          BUG_ON(proc->buffers.next == &buffer->entry);
09          prev = list_entry(buffer->entry.prev, struct binder_buffer, entry);
10          BUG_ON(!prev->free);
11          if (buffer_end_page(prev) == buffer_start_page(buffer)) {
12                  free_page_start = 0;
13                  if (buffer_end_page(prev) == buffer_end_page(buffer))
14                          free_page_end = 0;
15                  ......
16          }
17
18          if (!list_is_last(&buffer->entry, &proc->buffers)) {
19                  next = list_entry(buffer->entry.next,
20                                    struct binder_buffer, entry);
21                  if (buffer_start_page(next) == buffer_end_page(buffer)) {
22                          free_page_end = 0;
23                          if (buffer_start_page(next) ==
24                              buffer_start_page(buffer))
25                                  free_page_start = 0;
26                          ......
27                  }
28          }
29          list_del(&buffer->entry);
30          if (free_page_start || free_page_end) {
31                  ......
32                  binder_update_page_range(proc, 0, free_page_start ?
33                          buffer_start_page(buffer) : buffer_end_page(buffer),
34                          (free_page_end ? buffer_end_page(buffer) :
35                          buffer_start_page(buffer)) + PAGE_SIZE, NULL);
36          }
37  }
```

注意

在调用函数binder_delete_free_buffer来删除一个binder_buffer结构体buffer时，必须保证它指向的内核缓冲区不是目标进程proc的第一个内核缓冲区，并且该内核缓冲区以及它前面的内核缓冲区都是空闲的；否则，函数就会报错。在进一步描述函数binder_delete_free_buffer的实现之前，我们假设binder_buffer结构体prev和next分别指向要删除的内核缓冲区的前面一个内核缓冲区和后面一个内核缓冲区。

为了方便描述，我们假设binder_buffer结构体buffer、prev和next都是横跨在两个虚拟地址页面上的。第5行和第6行定义了两个变量free_page_start和free_page_end，并且将它们的值都设置为1，表示要释放binder_buffer结构体buffer所横跨的第一个虚拟地址页面和第二个虚拟地址页面所对应的物理页面。接下来会根据实际情况来调整变量free_page_start和free_page_end的值，以便可以正确地删除binder_buffer结构体buffer。

第11行到第16行的if语句块检查binder_buffer结构体buffer和prev的位置关系。如果binder_buffer结构体buffer的第一个虚拟地址页面和binder_buffer结构体prev的第二个虚拟地址页面是同一个页面，即第11行的if语句为true，那么binder_buffer结构体buffer所在的第一个虚拟地址页面所对应的物理页面就不可以释放，因此，第12行将变量free_page_start的值设置为0。如果binder_buffer结构体buffer的第二个虚拟地址页面和binder_buffer结构体prev的第二个虚拟地址页面也是同一个页面，即第13行的if语句为true，那么binder_buffer结构体buffer所在的第二个虚拟地址页面所对应的物理页面也不可以释放，因此，第14行也将变量free_page_end的值设置为0。这两种情况表明binder_buffer结构体buffer和prev同时位于一个虚拟地址页面中，如图5-13所示。

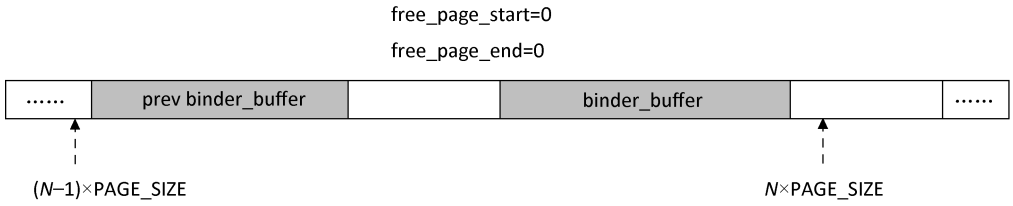


图5–13　要删除的binder_buffer结构体和前一个binder_buffer结构体位于同一个虚拟地址页面

第21行到第27行的if语句块检查binder_buffer结构体buffer和next的位置关系。如果binder_buffer结构体buffer的第二个虚拟地址页面和binder_buffer结构体next的第一个虚拟地址页面是同一个页面，即第21行的if语句为true，那么binder_buffer结构体buffer所在的第二个虚拟地址页面所对应的物理页面就不可以释放，因此，第22行将变量free_page_end的值设置为0。如果binder_buffer结构体buffer的第一个虚拟地址页面和binder_buffer结构体next的第一个虚拟地址页面也是同一个页面，即第23行的if语句为true，那么binder_buffer结构体buffer所在的第一个虚拟地址页面所对应的物理页面也不可以释放，因此，第25行也将变量free_page_start的值设置为0。这两种情况表明binder_buffer结构体buffer和next同时位于一个虚拟地址页面中，如图5–14所示。

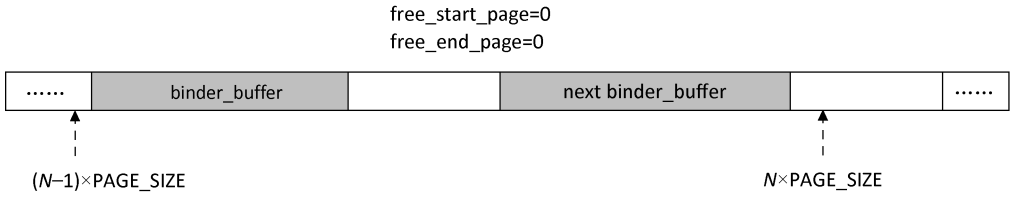


图5–14　要删除的binder_buffer结构体和后一个binder_buffer结构体位于同一个虚拟地址页面

调整好变量free_page_start和free_page_end的值后，第29行首先将binder_buffer结构体buffer所描述的内核缓冲区从目标进程proc的内核缓冲区列表中删除，接着第30行的if语句检查是否需要释放binder_buffer结构体buffer所占用的物理页面。如果变量free_page_start或者free_page_end的值有一个等于1，那么第32行就需要调用函数binder_update_page_range来释放它所在的虚拟地址页面所对应的物理页面。变量free_page_start、free_page_end的值与要释放的物理页面的对应关系如表5–1所示。

表5–1 变量free_page_start、free_page_end的值与要释放的物理页面的对应关系

free_page_end \ free_page_start	1	0
1	释放两个物理页面 buffer_start_page(buffer) ~ buffer_end_page(buffer) + PAGE_SIZE	释放第二个物理页面 buffer_end_page(buffer) ~ buffer_end_page(buffer) + PAGE_SIZE
0	释放第一个物理页面 buffer_start_page(buffer) ~ buffer_start_page(buffer) + PAGE_SIZE	不用释放物理页面

5.1.5.3　查询内核缓冲区

在前面的5.1.5.2小节中提到，当一个进程使用完成一个内核缓冲区之后，它就会主动使用命令协议BC_FREE_BUFFER来通知Binder驱动程序释放内核缓冲区所对应的物理页面。然而，进程只知道要释放的内核缓冲区的用户空间地址，而Binder驱动程序需要知道用来描述该内核缓冲区的一个binder_buffer结

构体，然后才可以释放它所占用的物理页面。因此，Binder驱动程序提供了函数binder_buffer_lookup根据一个用户空间地址来查询一个内核缓冲区，它的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static struct binder_buffer *binder_buffer_lookup(
02         struct binder_proc *proc, void __user *user_ptr)
03 {
04         struct rb_node *n = proc->allocated_buffers.rb_node;
05         struct binder_buffer *buffer;
06         struct binder_buffer *kern_ptr;
07
08         kern_ptr = user_ptr - proc->user_buffer_offset
09                 - offsetof(struct binder_buffer, data);
10
11         while (n) {
12                 buffer = rb_entry(n, struct binder_buffer, rb_node);
13                 BUG_ON(buffer->free);
14
15                 if (kern_ptr < buffer)
16                         n = n->rb_left;
17                 else if (kern_ptr > buffer)
18                         n = n->rb_right;
19                 else
20                         return buffer;
21         }
22         return NULL;
23 }
```

Binder驱动程序将一个内核缓冲区的数据传递给目标进程时，它只把数据缓冲区的用户地址传递给目标进程，即将一个binder_buffer结构体的成员变量data所指向的一块数据缓冲区的用户地址传递给目标进程。了解了这个背景知识之后，我们就可以知道，参数user_ptr是一个用户空间地址，并且它指向一个binder_buffer结构体的成员变量data的地址。

第8行和第9行中的子表达式user_ptr – offsetof(struct binder_buffer, data)用来计算一个binder_buffer结构体的用户空间地址，接着再减去目标进程proc的成员变量user_buffer_offset，就得到用户空间地址user_ptr所对应的binder_buffer结构体的内核空间地址了，最后将它保存在变量kern_ptr中。

前面计算得到的binder_buffer结构体的内核空间地址不一定指向一个有效的内核缓冲区，因此，函数就需要对它进行验证。只有验证通过后，才可以将它返回给调用者。在前面的5.1.1小节中介绍进程结构体binder_proc时提到，一个进程的已经分配物理页面的内核缓冲区，是以它的内核空间地址作为关键字保存在进程的已分配内核缓冲区红黑树allocated_buffers中的，因此，如果函数能够根据前面计算得到的binder_buffer结构体的内核空间地址在这个红黑树中找到一个对应的节点，那么就说明该binder_buffer结构体是指向一个有效的内核缓冲区的。第11行到第21行的while循环便是在目标进程proc的已分配内核缓冲区红黑树allocated_buffers中查找与内核空间地址kern_ptr对应的节点。如果能找到，那么第20行就将该节点所对应的一个binder_buffer结构体返回给调用者；否则，第22行就返回一个NULL值给调用者，表示找不到与用户空间地址user_ptr对应的内核缓冲区。

至此，我们就分析完成Binder驱动程序的基础知识了。接下来，我们继续分析应用程序框架层中的Binder进程间通信库的实现。

5.2　Binder进程间通信库

Android系统在应用程序框架层中将各种Binder驱动程序操作封装成一个Binder库，这样进程就可以方便地调用Binder库提供的接口来实现进程间通信。在本节中，我们将学习Binder库的基础知识，

以便加深对Binder驱动程序的理解。在接下来的5.3小节中，我们将使用Binder库创建一个进程间通信应用实例。

在Binder库中，Service组件和Client组件分别使用模板类BnInterface和BpInterface来描述，其中，前者称为Binder本地对象，后者称为Binder代理对象。Binder库中的Binder本地对象和Binder代理对象分别对应于Binder驱动程序中的Binder实体对象和Binder引用对象。下面我们就主要介绍这两个模板类的实现。

模板类BnInterface和BpInterface的定义如下所示。

frameworks/base/include/binder/IInterface.h

```
01 template<typename INTERFACE>
02 class BnInterface : public INTERFACE, public BBinder
03 {
04 public:
05     virtual sp<IInterface>      queryLocalInterface(const String16& _descriptor);
06     virtual const String16&     getInterfaceDescriptor() const;
07
08 protected:
09     virtual IBinder*            onAsBinder();
10 };
11
12 template<typename INTERFACE>
13 class BpInterface : public INTERFACE, public BpRefBase
14 {
15 public:
16                                 BpInterface(const sp<IBinder>& remote);
17
18 protected:
19     virtual IBinder*            onAsBinder();
20 };
```

模板参数INTERFACE是一个由进程自定义的Service组件接口，模板类BnInterface和BpInterface都需要实现该接口。在使用Binder库开发Service组件和Client组件时，除了要定义Service组件接口之外，还必须要实现一个Binder本地对象类和一个Binder代理对象类，它们分别继承于模板类BnInterface和BpInterface。例如，在接下来的5.3小节要介绍的Binder进程间通信应用实例中，我们首先定义一个硬件访问服务接口IFregService，用来访问第2章介绍的虚拟硬件设备freg的寄存器val的值；然后又自定义一个Binder本地对象类BnFregService和一个Binder代理对象类BpFregService，它们分别继承于模板类BnInterface和BpInterface，并且实现了IFregService接口。

模板类BnInterface继承了BBinder类，后者为Binder本地对象提供了抽象的进程间通信接口，它的定义如下所示。

frameworks/base/include/binder/Binder.h

```
01 class BBinder : public IBinder
02 {
03 public:
04     ......
05
06     virtual status_t    transact(   uint32_t code,
07                                     const Parcel& data,
08                                     Parcel* reply,
09                                     uint32_t flags = 0);
10
11     ......
12
13 protected:
14     ......
15
```

```
16     virtual status_t    onTransact( uint32_t code,
17                                     const Parcel& data,
18                                     Parcel* reply,
19                                     uint32_t flags = 0);
20
21     ......
22 };
```

BBinder类有两个重要的成员函数transact和onTransact。当一个Binder代理对象通过Binder驱动程序向一个Binder本地对象发出一个进程间通信请求时，Binder驱动程序就会调用该Binder本地对象的成员函数transact来处理该请求。成员函数onTransact是由BBinder的子类，即Binder本地对象类来实现的，它负责分发与业务相关的进程间通信请求。事实上，与业务相关的进程间通信请求是由Binder本地对象类的子类，即Service组件类来负责处理的。例如，在接下来的5.3小节将要介绍的Binder进程间通信应用实例中，Binder本地对象类BnFregService重写了父类BBinder的成员函数onTransact，并且将访问虚拟硬件设备freg的寄存器val的操作分发给其子类FregService来处理。

BBinder类又继承了IBinder类，而后者又继承了RefBase类。从第3章的内容可以知道，继承了RefBase类的子类的对象均可以通过强指针和弱指针来维护它们的生命周期，换句话说，就是Binder本地对象是通过引用计数技术来维护生命周期的。由于Binder本地对象受到Binder驱动程序中的Binder实体对象，以及运行在Client进程中的Binder代理对象引用，因此，它的引用计数技术比较复杂，在后面的5.4小节中，我们再详细分析。

模板类BpInterface继承了BpRefBase类，后者为Binder代理对象提供了抽象的进程间通信接口，它的定义如下所示。

frameworks/base/include/binder/Binder.h

```
01 class BpRefBase : public virtual RefBase
02 {
03 protected:
04                             BpRefBase(const sp<IBinder>& o);
05     ......
06
07     inline  IBinder*        remote()                { return mRemote; }
08     inline  IBinder*        remote() const          { return mRemote; }
09
10 private:
11     ......
12
13     IBinder* const          mRemote;
14     ......
15 };
```

BpRefBase类又继承了RefBase类，因此，它的子类对象，即Binder代理对象也可以通过强指针和弱指针来维护生命周期。同样，在后面的5.4小节中，我们再详细分析Binder代理对象的生命周期。

BpRefBase类有一个重要的成员变量mRemote，它指向一个BpBinder对象，可以通过成员函数remote来获取。BpBinder类实现了BpRefBase类的进程间通信接口，它的定义如下所示。

frameworks/base/include/binder/BpBinder.h

```
01 class BpBinder : public IBinder
02 {
03 public:
04                         BpBinder(int32_t handle);
05
06     inline  int32_t     handle() const { return mHandle; }
07
08     ......
09
10     virtual status_t    transact(   uint32_t code,
```

```
11                                      const Parcel& data,
12                                      Parcel* reply,
13                                      uint32_t flags = 0);
14
15     ......
16
17 private:
18     const   int32_t             mHandle;
19
20     ......
21 };
```

BpBinder类的成员变量mHandle是一个整数，它表示一个Client组件的句柄值，可以通过成员函数handle来获取。在前面的5.1.1小节中介绍结构体binder_ref时提到，每一个Client组件在Binder驱动程序中都对应有一个Binder引用对象，而每一个Binder引用对象都有一个句柄值，其中，Client组件就是通过这个句柄值来和Binder驱动程序中的Binder引用对象建立对应关系的。

BpBinder类的成员函数transact用来向运行在Server进程中的Service组件发送进程间通信请求，这是通过Binder驱动程序间接实现的。BpBinder类的成员函数transact会把BpBinder类的成员变量mHandle，以及进程间通信数据发送给Binder驱动程序，这样Binder驱动程序就能够根据这个句柄值来找到对应的Binder引用对象，继而找到对应的Binder实体对象，最后就可以将进程间通信数据发送给对应的Service组件了。

无论是BBinder类，还是BpBinder类，它们都是通过IPCThreadState类来和Binder驱动程序交互的。IPCThreadState类的定义如下所示。

frameworks/base/include/binder/IPCThreadState.h

```
01 class IPCThreadState
02 {
03 public:
04     static  IPCThreadState*     self();
05
06     ......
07
08             status_t            transact(int32_t handle,
09                                          uint32_t code, const Parcel& data,
10                                          Parcel* reply, uint32_t flags);
11
12     ......
13
14 private:
15             status_t            talkWithDriver(bool doReceive=true);
16     ......
17
18     const   sp<ProcessState>    mProcess;
19
20     ......
21 };
```

在前面的5.1.1小节中介绍结构体binder_proc时提到，每一个使用了Binder进程间通信机制的进程都有一个Binder线程池，用来处理进程间通信请求。对于每一个Binder线程来说，它的内部都有一个IPCThreadState对象，我们可以通过IPCThreadState类的静态成员函数self来获取，并且调用它的成员函数transanct来和Binder驱动程序交互。在IPCThreadState类的成员函数transanct内部，与Binder驱动程序的交互操作又是通过调用成员函数talkWithDriver来实现的，它一方面负责向Binder驱动程序发送进程间通信请求，另一方面又负责接收来自Binder驱动程序的进程间通信请求。

IPCThreadState类有一个成员变量mProcess，它指向一个ProcessState对象。对于每一个使用了Binder进程间通信机制的进程来说，它的内部都有一个ProcessState对象，它负责初始化Binder设备，

即打开设备文件/dev/binder，以及将设备文件/dev/binder映射到进程的地址空间。由于这个ProcessState对象在进程范围内是唯一的，因此，Binder线程池中的每一个线程都可以通过它来和Binder驱动程序建立连接。

ProcessState类的定义如下所示。

frameworks/base/include/binder/ ProcessState.h

```
01 class ProcessState : public virtual RefBase
02 {
03 public:
04     static  sp<ProcessState>    self();
05     ......
06
07 private:
08     ......
09             int                 mDriverFD;
10             void*               mVMStart;
11
12     ......
13 };
```

进程中的ProcessState对象可以通过ProcessState类的静态成员函数self来获取。第一次调用ProcessState类的静态成员函数self时，Binder库就会为进程创建一个ProcessState对象，并且调用函数open来打开设备文件/dev/binder，接着又调用函数mmap将它映射到进程的地址空间，即请求Binder驱动程序为进程分配内核缓冲区。设备文件/dev/binder映射到进程的地址空间后，得到的内核缓冲区的用户地址就保存在其成员变量mVMStart中。

至此，Binder库的基础知识就介绍完了。下面我们通过图5-15和图5-16来总结Service组件和Client组件的实现原理。

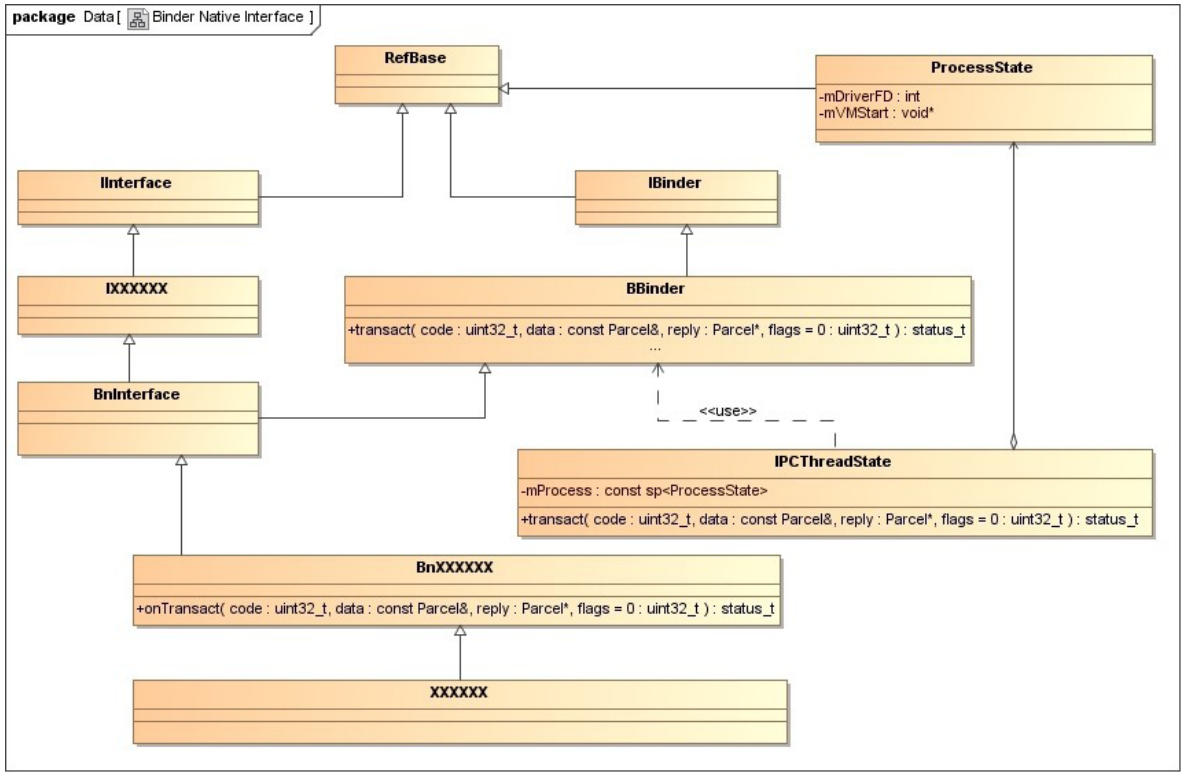


图5-15　Service组件的实现原理

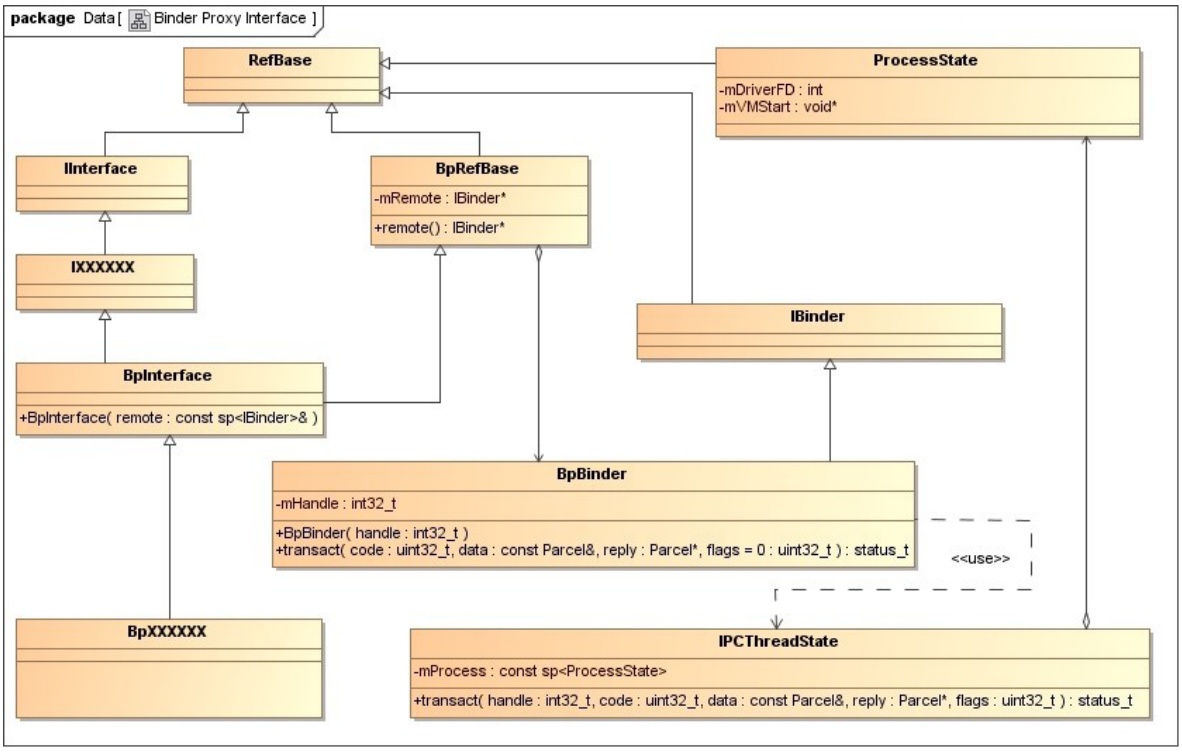


图5-16　Client组件的实现原理

在图5-15和图5-16中，IXXXXXX表示Service组件和Client组件都要实现的服务接口；BnXXXXXX、BpXXXXXX分别表示Service组件和Client组件要继承的Binder本地对象类和Binder代理对象类；XXXXXX表示一个Service组件。在接下来的5.3小节要介绍的Binder进程间通信应用实例中，这四个类就分别为IFregService、BnFregService、BpFregService和FregService。实现了这四个类，我们就可以使用Binder库来实现进程间通信了。

5.3　Binder进程间通信应用实例

本节将基于应用程序框架层提供的Binder库来开发一个Binder进程间通信应用实例，它包含一个Server进程和一个Client进程。其中，Server进程实现了一个Service组件，负责管理第2章所介绍的虚拟硬件设备freg的寄存器val，并且向Client进程提供访问服务。

我们将该应用实例划分为：common、server和client三个模块。其中，模块common实现了硬件访问服务接口IFregService，以及Binder本地对象类BnFregService和Binder代理对象类BpFregService；模块server实现了一个Server进程，它里面包含有一个Service组件FregService；模块client实现了一个Client进程，它通过一个BpFregService代理对象去访问运行在Server进程中的Service组件FregService所提供的服务。

应用实例的目录结构如下：

```
~/Android/external/binder
----common
     ----IFregService.h
     ----IFregService.cpp
----server
     ----FregServer.cpp
     ----Android.mk
```

```
----client
     ----FregClient.cpp
     ----Android.mk
```

每一个模块都包含了两个文件。接下来，我们详细分析每一个文件的内容。

common/IFregService.h

```
01 #ifndef IFREGSERVICE_H_
02 #define IFREGSERVICE_H_
03
04 #include <utils/RefBase.h>
05 #include <binder/IInterface.h>
06 #include <binder/Parcel.h>
07
08 #define FREG_SERVICE "shy.luo.FregService"
09
10 using namespace android;
11
12 class IFregService: public IInterface
13 {
14 public:
15         DECLARE_META_INTERFACE(FregService);
16         virtual int32_t getVal() = 0;
17         virtual void setVal(int32_t val) = 0;
18 };
19
20 class BnFregService: public BnInterface<IFregService>
21 {
22 public:
23         virtual status_t onTransact(uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags = 0);
24 };
25
26 #endif
```

第8行定义的宏FREG_SERVICE用来描述Service组件FregService注册到Service Manager的名称。第12行到第18行定义了一个硬件访问服务接口IFregService，它有两个成员函数getVal和setVal，分别用来读取和写入虚拟硬件设备freg中的寄存器val的值。第20行到第24行定义了一个Binder本地对象类BnFregService，它实现了模板类BnInterface的成员函数onTransact。

第15行使用宏DECLARE_META_INTERFACE来声明IFregService类的元接口，如下所示。

frameworks/base/include/binder/IInterface.h

```
1 #define DECLARE_META_INTERFACE(INTERFACE)                               \
2     static const android::String16 descriptor;                        \
3     static android::sp<I##INTERFACE> asInterface(                     \
4             const android::sp<android::IBinder>& obj);                \
5     virtual const android::String16& getInterfaceDescriptor() const;  \
6     I##INTERFACE();                                                   \
7     virtual ~I##INTERFACE();                                          \
```

宏DECLARE_META_INTERFACE展开之后，就得到以下的代码。

```
1     static const android::String16 descriptor;
2     static android::sp<IFregService> asInterface(
3             const android::sp<android::IBinder>& obj);
4     virtual const android::String16& getInterfaceDescriptor() const;
5     IFregService();
6     virtual ~IFregService();
```

它为IFregService类定义了一个静态成员变量descriptor，用来描述接口的名称，可以通过成员函数getInterfaceDescriptor来获取。同时，还定义了一个静态成员函数asInterface，用来将一个IBinder对象转

换为一个IFregService接口。最后，还定义了IFregService类的构造函数和析构函数。

common/IFregService.cpp

```
01 #define LOG_TAG "IFregService"
02
03 #include <utils/Log.h>
04
05 #include "IFregService.h"
06
07 using namespace android;
08
09 enum
10 {
11         GET_VAL = IBinder::FIRST_CALL_TRANSACTION,
12         SET_VAL
13 };
14
15 class BpFregService: public BpInterface<IFregService>
16 {
17 public:
18         BpFregService(const sp<IBinder>& impl)
19                 : BpInterface<IFregService>(impl)
20         {
21
22         }
23
24 public:
25         int32_t getVal()
26         {
27                 Parcel data;
28                 data.writeInterfaceToken(IFregService::getInterfaceDescriptor());
29
30                 Parcel reply;
31                 remote()->transact(GET_VAL, data, &reply);
32
33                 int32_t val = reply.readInt32();
34
35                 return val;
36         }
37
38         void setVal(int32_t val)
39         {
40                 Parcel data;
41                 data.writeInterfaceToken(IFregService::getInterfaceDescriptor());
42                 data.writeInt32(val);
43
44                 Parcel reply;
45                 remote()->transact(SET_VAL, data, &reply);
46         }
47
48 };
49
50 IMPLEMENT_META_INTERFACE(FregService, "shy.luo.IFregService");
51
52 status_t BnFregService::onTransact(uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
53 {
54         switch(code)
55         {
56                 case GET_VAL:
57                 {
58                         CHECK_INTERFACE(IFregService, data, reply);
59
```

```
60                          int32_t val = getVal();
61                          reply->writeInt32(val);
62
63                          return NO_ERROR;
64                  }
65                  case SET_VAL:
66                  {
67                          CHECK_INTERFACE(IFregService, data, reply);
68
69                          int32_t val = data.readInt32();
70                          setVal(val);
71
72                          return NO_ERROR;
73                  }
74                  default:
75                  {
76                          return BBinder::onTransact(code, data, reply, flags);
77                  }
78          }
79 }
```

第11行和第12行定义了两个进程间通信代码GET_VAL和SET_VAL，它们分别对应于IFregService接口中的两个成员函数getVal和setVal。

第15行到第48行定义了一个Binder代理对象类BpFregService，它继承了模块类BpInterface，并且实现了IFregService接口。在它的成员函数getVal中，首先将要传递的数据封装在一个Parcel对象data中，然后调用父类BpRefBase的成员函数remote来获得一个BpBinder代理对象，最后调用这个BpBinder代理对象的成员函数transact来请求运行在Server进程中的一个Binder本地对象执行一个GET_VAL操作。该GET_VAL操作的返回结果是一个整数，封装在另外一个Parcel对象reply中，表示虚拟硬件设备freg的寄存器val的值。在它的成员函数setVal中，同样是首先将要传递的数据封装在一个Parcel对象data中，然后使用其父类内部的一个BpBinder代理对象的成员函数transact来请求运行在Server进程中的一个Binder本地对象执行一个SET_VAL操作，该SET_VAL操作将一个整数写入到虚拟硬件设备freg的寄存器val中。

第50行使用宏IMPLEMENT_META_INTERFACE来实现IFregService类的元接口，它是与宏DECLARE_META_INTERFACE相对应的，如下所示。

frameworks/base/include/binder/IInterface.h

```
01 #define IMPLEMENT_META_INTERFACE(INTERFACE, NAME)                              \
02     const android::String16 I##INTERFACE::descriptor(NAME);                    \
03     const android::String16&                                                   \
04             I##INTERFACE::getInterfaceDescriptor() const {                     \
05         return I##INTERFACE::descriptor;                                       \
06     }                                                                          \
07     android::sp<I##INTERFACE> I##INTERFACE::asInterface(                       \
08             const android::sp<android::IBinder>& obj)                          \
09     {                                                                          \
10         android::sp<I##INTERFACE> intr;                                        \
11         if (obj != NULL) {                                                     \
12             intr = static_cast<I##INTERFACE*>(                                 \
13                 obj->queryLocalInterface(                                      \
14                         I##INTERFACE::descriptor).get());                      \
15             if (intr == NULL) {                                                \
16                 intr = new Bp##INTERFACE(obj);                                 \
17             }                                                                  \
18         }                                                                      \
19         return intr;                                                           \
20     }                                                                          \
21     I##INTERFACE::I##INTERFACE() { }                                           \
22     I##INTERFACE::~I##INTERFACE() { }                                          \
```

宏IMPLEMENT_META_INTERFACE展开之后，就得到以下的代码。

```
01     const android::String16 IFregService::descriptor("shy.luo.IFregService");
02     const android::String16&
03             IFregService::getInterfaceDescriptor() const {
04         return IFregService::descriptor;
05     }
06     android::sp<IFregService> IFregService::asInterface(
07             const android::sp<android::IBinder>& obj)
08     {
09         android::sp<IFregService> intr;
10         if (obj != NULL) {
11             intr = static_cast<IFregService*>(
12                 obj->queryLocalInterface(
13                         IFregService::descriptor).get());
14             if (intr == NULL) {
15                 intr = new BpFregService(obj);
16             }
17         }
18         return intr;
19     }
20     IFregService::IFregService() { }
21     IFregService::~IFregService() { }
```

它将IFregService类的静态成员变量descriptor设置为“shy.luo.IFregService”，并且实现了IFregService类的构造函数、析构函数，以及成员函数getInterfaceDescriptor和asInterface。IFregService类的构造函数和析构函数是一个空实现，而成员函数getInterfaceDescriptor用来获取IFregService类的描述符，即获取静态成员变量descriptor的值，最后成员函数asInterface用来将一个IBinder对象转换为一个IFregService接口。

IFregService类的成员函数asInterface的参数obj应该指向一个类型为BnFregService的Binder本地对象，或者一个类型为BpBinder的Binder代理对象；否则，它的返回值就会等于NULL。如果参数指向的是一个BnFregService对象，那么第11行到第13行调用它的成员函数queryLocalInterface就可以直接返回一个IFregService接口；如果参数obj指向的是一个BpBinder代理对象，那么它的成员函数queryLocalInterface的返回值就为NULL，因此，接着第15行就会将该BpBinder代理对象封装成一个BpFregService对象，并且将它的IFregService接口返回给调用者。

回到IFregService.cpp文件中，第52行到第79行定义了BnFregService类的成员函数onTransact。BnFregService类的成员函数onTransact负责将GET_VAL和SET_VAL进程间通信请求分发给其子类的成员函数getVal和setVal来处理。BnFregService类的子类为FregService，它的成员函数getVal和setVal分别用来读取和写入虚件硬件设备freg的寄存器val的值。

注意

BnFregService类的成员函数onTransact在将GET_VAL和SET_VAL进程间通信请求分发给其子类处理之前，会首先调用宏CHECK_INTERFACE来检查该进程间通信请求的合法性，即检查该请求是否是由FregService组件的代理对象发送过来的。如果是，那么传递过来的Parcel对象data中的第一个数据应该是一个IFregService接口描述符，即“shy.luo.IFregService”；如果不是，那么BnFregService类的成员函数onTransact就会认为这是一个非法的进程间通信请求，因此，就不会继续向下执行了。

server/FregServer.cpp

```
01 #define LOG_TAG "FregServer"
02
03 #include <stdlib.h>
```

```
#include <fcntl.h>

#include <utils/Log.h>
#include <binder/IServiceManager.h>
#include <binder/IPCThreadState.h>

#include "../common/IFregService.h"

#define FREG_DEVICE_NAME "/dev/freg"

class FregService : public BnFregService
{
public:
        FregService()
        {
                fd = open(FREG_DEVICE_NAME, O_RDWR);
                if(fd == -1) {
                        LOGE("Failed to open device %s.\n", FREG_DEVICE_NAME);
                }
        }

        virtual ~FregService()
        {
                if(fd != -1) {
                        close(fd);
                }
        }

public:
        static void instantiate()
        {
                defaultServiceManager()->addService(String16(FREG_SERVICE), new FregService());
        }

        int32_t getVal()
        {
                int32_t val = 0;

                if(fd != -1) {
                        read(fd, &val, sizeof(val));
                }

                return val;
        }

        void setVal(int32_t val)
        {
                if(fd != -1) {
                        write(fd, &val, sizeof(val));
                }
        }

private:
        int fd;
};

int main(int argc, char** argv)
{
        FregService::instantiate();

        ProcessState::self()->startThreadPool();
        IPCThreadState::self()->joinThreadPool();

        return 0;
}
```

这是server模块的源代码文件。第14行到第58行实现了一个Service组件类FregService，它继承了BnFregService类，并且实现了IFregService接口。在FregService类的构造函数中，会调用函数open来打开设备文件/dev/freg，并且将得到的文件描述符保存在成员变量fd中。相应地，在FregService类的析构函数中，会调用函数close来关闭设备文件/dev/freg。打开设备文件/dev/freg之后，成员函数getVal和setVal就可以读取和写入虚拟硬件设备freg的寄存器val的值了。

FregService类的静态成员函数instantiate负责将一个FregService组件注册到Service Manager中，并且将它的注册名称设置为“shy.luo.FregService”。这样，Client进程就可以通过名称“shy.luo.FregService”来获取这个FregService组件的一个代理对象了。

第60行到第68行定义了server模块的入口函数main。在main函数中，第62行首先调用FregService类的静态成员函数instantiate 将一个FregService组件注册到Service Manager中，接着第64行调用进程中的ProcessState对象的成员函数startThreadPool来启动一个Binder线程池，最后第65行调用主线程的IPCThreadState对象的成员函数joinThreadPool将主线程添加到进程的Binder线程池中，用来处理来自Client进程的通信请求。

server/Android.mk

```
1 LOCAL_PATH := $(call my-dir)
2 include $(CLEAR_VARS)
3 LOCAL_MODULE_TAGS := optional
4 LOCAL_SRC_FILES := ../common/IFregService.cpp \
5         FregServer.cpp
6 LOCAL_SHARED_LIBRARIES:= libcutils libutils libbinder
7 LOCAL_MODULE := FregServer
8 include $(BUILD_EXECUTABLE)
```

这是server模块的编译脚本文件，它会在out/target/product/gerneric/system/bin目录中生成一个可执行文件FregServer。

client/FregClient.cpp

```
01 #define LOG_TAG "FregClient"
02
03 #include <utils/Log.h>
04 #include <binder/IServiceManager.h>
05
06 #include "../common/IFregService.h"
07
08 int main()
09 {
10         sp<IBinder> binder = defaultServiceManager()->getService(String16(FREG_SERVICE));
11         if(binder == NULL) {
12                 LOGE("Failed to get freg service: %s.\n", FREG_SERVICE);
13                 return -1;
14         }
15
16         sp<IFregService> service = IFregService::asInterface(binder);
17         if(service == NULL) {
18                 LOGE("Failed to get freg service interface.\n");
19                 return -2;
20         }
21
22         printf("Read original value from FregService:\n");
23
24         int32_t val = service->getVal();
25         printf(" %d.\n", val);
26
```

```
27          printf("Add value 1 to FregService.\n");
28
29          val += 1;
30          service->setVal(val);
31
32          printf("Read the value from FregService again:\n");
33
34          val = service->getVal();
35          printf(" %d.\n", val);
36
37          return 0;
38  }
```

这是client模块的源代码文件，它定义了client模块的入口函数main。

第10行首先调用函数defaultServiceManager 来获得Service Manager的一个代理对象，接着再调用它的成员函数getService来获得一个名称为“shy.luo.FregService”的Service组件的一个类型为BpBinder的代理对象。从server模块的实现可以知道，名称为“shy.luo.FregService”的Service组件正好是前面注册的一个FregService组件，因此，第16行就将前面获得的BpBinder代理对象封装成为一个BpFregService代理对象，并且取得它的IFregService接口，保存在变量service中。

第24行调用IFregService接口service的成员函数getVal从运行在另外一个进程中的FregService组件获取虚拟硬件设备freg的寄存器val的值，接着第25行将它打印出来。第29行和第30行调用IFregService接口service的成员函数setVal来请求运行在另外一个进程中的FregService组件将虚拟硬件设备freg的寄存器val的值加上1。最后，第34行再次调用IFregService的成员函数getVal来获取虚拟硬件设备freg的寄存器val的值，接着第35行将它打印出来。

从前面2.1小节的内容可以知道，虚拟硬件设备freg的寄存器val的值被初始化为0，因此，如果一切正常的话，第一次运行client 模块时，输出的两个整数值应该分别为0和1。

client/Android.mk

```
1 LOCAL_PATH := $(call my-dir)
2 include $(CLEAR_VARS)
3 LOCAL_MODULE_TAGS := optional
4 LOCAL_SRC_FILES := ../common/IFregService.cpp \
5          FregClient.cpp
6 LOCAL_SHARED_LIBRARIES:= libcutils libutils libbinder
7 LOCAL_MODULE := FregClient
8 include $(BUILD_EXECUTABLE)
```

这是client模块的编译脚本文件，它会在out/target/product/gerneric/system/bin目录中生成一个可执行文件FregClient。

模块common、server和client的源文件，以及编译脚本文件都准备好之后，我们就可以对它们进行编译和打包了。

```
USER@MACHINE:~/Android$ mmm ./external/binder/server/
USER@MACHINE:~/Android$ mmm ./external/binder/client/
USER@MACHINE:~/Android$ make snod
```

最后，我们使用新的Android系统镜像文件system.img来启动Android模拟器，接着使用adb工具连接上它，并且进入到/system/bin目录中，分别将应用程序FregServer和FregClient运行起来，如下所示。

```
USER@MACHINE: ~/Android$ emulator -kernel kernel/goldfish/arch/arm/boot/zImage &
USER@MACHINE: ~/Android$ adb shell
root@android:/ # cd system/bin
root@android:/system/bin # ./FregServer &
root@android:/system/bin # ./FregClient
Read original value from FregService:
```

```
0.
Add value 1 to FregService.
Read the value from FregService again:
1.
```

如果能够看到上面的输出，那么就说明应用程序FregClient和应用程序FregServer能够成功地使用Binder库来执行进程间通信了。

通过对Binder库及其应用实例的学习，我们就对Binder进程间通信机制有了一个感性的认识。接下来，我们再进一步来学习Binder进程间通信机制的两个关键技术，即Binder对象的引用计数技术和死亡通知机制。

5.4 Binder对象引用计数技术

通过对Binder驱动程序和Binder库的学习，我们知道，在Client进程和Server进程的一次通信过程中，涉及了四种类型的对象，它们分别是位于Binder驱动程序中的Binder实体对象（binder_node）和Binder引用对象（binder_ref），以及位于Binder库中的Binder本地对象（BBinder）和Binder代理对象（BpBinder），它们的交互过程如图5-17所示。

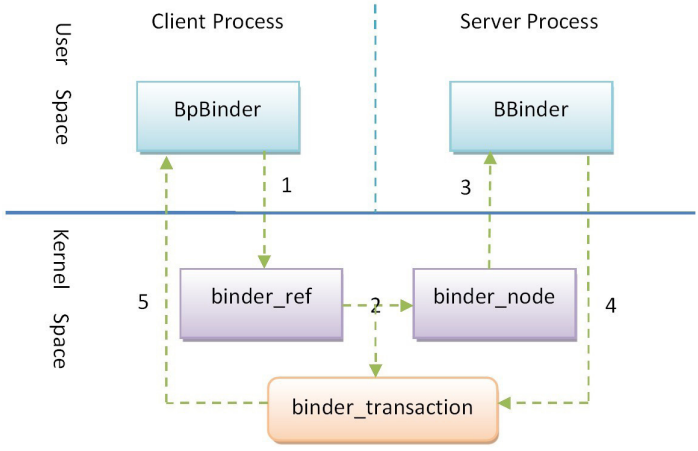


图5-17 Biner实体对象、Binder引用对象、Binder本地对象和Binder代理对象的交互过程

它们的交互过程可以划分为五个步骤，如下所示。

（1）运行在Client进程中的Binder代理对象通过Binder驱动程序向运行在Server进程中的Binder本地对象发出一个进程间通信请求，Binder驱动程序接着就根据Client进程传递过来的Binder代理对象的句柄值来找到对应的Binder引用对象。

（2）Binder驱动程序根据前面找到的Binder引用对象找到对应的Binder实体对象，并且创建一个事务（binder_transaction）来描述该次进程间通信过程。

（3）Binder驱动程序根据前面找到的Binder实体对象来找到运行在Server进程中的Binder本地对象，并且将Client进程传递过来的通信数据发送给它处理。

（4）Binder本地对象处理完成Client进程的通信请求之后，就将通信结果返回给Binder驱动程序，Binder驱动程序接着就找到前面所创建的一个事务。

（5）Binder驱动程序根据前面找到的事务的相关属性来找到发出通信请求的Client进程，并且通知Client进程将通信结果返回给对应的Binder代理对象处理。

从这个过程就可以看出，Binder代理对象依赖于Binder引用对象，而Binder引用对象又依赖于Binder实体对象，最后，Binder实体对象又依赖于Binder本地对象。这样，Binder进程间通信机制就必

须采用一种技术措施来保证，不能销毁一个还被其他对象依赖着的对象。为了维护这些Binder对象的依赖关系，Binder进程间通信机制采用了引用计数技术来维护每一个Binder对象的生命周期。

接下来，我们就分析Binder驱动程序和Binder库是如何维护Binder本地对象、Binder实体对象、Binder引用对象和Binder代理对象的生命周期的。

5.4.1　Binder本地对象的生命周期

Binder本地对象是一个类型为BBinder的对象，它是在用户空间中创建的，并且运行在Server进程中。Binder本地对象一方面会被运行在Server进程中的其他对象引用，另一方面也会被Binder驱动程序中的Binder实体对象引用。由于BBinder类继承了RefBase类，因此，Server进程中的其他对象可以简单地通过智能指针来引用这些Binder本地对象，以便可以控制它们的生命周期。由于Binder驱动程序中的Binder实体对象是运行在内核空间的，它不能够通过智能指针来引用运行在用户空间的Binder本地对象，因此，Binder驱动程序就需要和Server进程约定一套规则来维护它们的引用计数，避免它们在还被Binder实体对象引用的情况下销毁。

Server进程将一个Binder本地对象注册到Server Manager时，Binder驱动程序就会为它创建一个Binder实体对象。接下来，当Client进程通过Service Manager来查询一个Binder本地对象的代理对象接口时，Binder驱动程序就会为它所对应的Binder实体对象创建一个Binder引用对象，接着再使用BR_INCREFS和BR_ACQUIRE协议来通知对应的Server进程增加对应的Binder本地对象的弱引用计数和强引用计数。这样就能保证Client进程中的Binder代理对象在引用一个Binder本地对象期间，该Binder本地对象不会被销毁。当没有任何Binder代理对象引用一个Binder本地对象时，Binder驱动程序就会使用BR_DECREFS和BR_RELEASE协议来通知对应的Server进程减少对应的Binder本地对象的弱引用计数和强引用计数。

总结来说，Binder驱动程序就是通过BR_INCREFS、BR_ACQUIRE、BR_DECREFS和BR_RELEASE协议来引用运行在Server进程中的Binder本地对象的，相关的代码实现在函数binder_thread_read中，如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static int
02 binder_thread_read(struct binder_proc *proc, struct binder_thread *thread,
03         void  __user *buffer, int size, signed long *consumed, int non_block)
04 {
05           ......
06
07           while (1) {
08                 uint32_t cmd;
09                 ......
10                 struct binder_work *w;
11                 ......
12
13                 if (!list_empty(&thread->todo))
14                         w = list_first_entry(&thread->todo, struct binder_work, entry);
15                 else if (!list_empty(&proc->todo) && wait_for_proc_work)
16                         w = list_first_entry(&proc->todo, struct binder_work, entry);
17                 else {
18                         ......
19                 }
20
21                     ......
22
23                     switch (w->type) {
24                     ......
```

```
25                      case BINDER_WORK_NODE: {
26                          struct binder_node *node = container_of(w, struct binder_node, work);
27                          uint32_t cmd = BR_NOOP;
28                          const char *cmd_name;
29                          int strong = node->internal_strong_refs || node->local_strong_refs;
30                          int weak = !hlist_empty(&node->refs) || node->local_weak_refs || strong;
31                          if (weak && !node->has_weak_ref) {
32                                  cmd = BR_INCREFS;
33                                  cmd_name = "BR_INCREFS";
34                                  node->has_weak_ref = 1;
35                                  node->pending_weak_ref = 1;
36                                  node->local_weak_refs++;
37                          } else if (strong && !node->has_strong_ref) {
38                                  cmd = BR_ACQUIRE;
39                                  cmd_name = "BR_ACQUIRE";
40                                  node->has_strong_ref = 1;
41                                  node->pending_strong_ref = 1;
42                                  node->local_strong_refs++;
43                          } else if (!strong && node->has_strong_ref) {
44                                  cmd = BR_RELEASE;
45                                  cmd_name = "BR_RELEASE";
46                                  node->has_strong_ref = 0;
47                          } else if (!weak && node->has_weak_ref) {
48                                  cmd = BR_DECREFS;
49                                  cmd_name = "BR_DECREFS";
50                                  node->has_weak_ref = 0;
51                          }
52                          if (cmd != BR_NOOP) {
53                                  if (put_user(cmd, (uint32_t __user *)ptr))
54                                          return -EFAULT;
55                                  ptr += sizeof(uint32_t);
56                                  if (put_user(node->ptr, (void * __user *)ptr))
57                                          return -EFAULT;
58                                  ptr += sizeof(void *);
59                                  if (put_user(node->cookie, (void * __user *)ptr))
60                                          return -EFAULT;
61                                  ptr += sizeof(void *);
62                                  ......
63                          } else {
64                                  list_del_init(&w->entry);
65                                  ......
66                          }
67                      } break;
68                         ......
69                  }
70          }
71
72          ......
73          return 0;
74  }
```

注意

当Binder驱动程序要和目标进程或者目标线程通信时，它就会把一个工作项加入到它的todo队列中。目标进程或者目标线程会不断地调用Binder驱动程序中的函数binder_thread_read来检查它的todo队列中有没有新的工作项。如果有，目标进程或者目标线程就会将它取出来，并且返回到用户空间去处理。第13行到第19行的if语句就是用来检查目标进程或者目标线程中的todo队列的，并且将里面的待处理工作项保存在变量w中。

当Binder驱动程序使用协议BR_INCREFS、BR_ACQUIRE、BR_DECREFS和BR_RELEASE来和

Server进程通信时，它就将一个类型为BINDER_WORK_NODE的工作项加入到该Server进程的todo队列中。类型为BINDER_WORK_NODE的工作项是与一个Binder实体对象相关的，表示要修改与该Binder实体对象所对应的Binder本地对象的引用计数。第29行和第30行检查该Binder实体对象是否有强引用计数和弱引用计数。如果该Binder实体对象有强引用计数，那么就将变量strong的值设置为1；否则，就设置为0。同样，如果该Binder实体对象有弱引用计数，那么就将变量weak的值设置为1；否则，就设置为0。

在前面的5.1.1小节中介绍结构体binder_node时提到，如果Binder驱动程序已经请求运行Server进程中的一个Binder本地对象为它内部的一个Binder实体对象增加了强引用计数和弱引用计数，那么Binder驱动程序就会将该Binder实体对象的成员变量has_strong_ref和has_weak_ref的值设置为1；否则，就设置为0，表示接下来要增加该Binder本地对象的强引用计数和弱引用计数。

第31行到第51行的if语句就根据变量strong和weak的值，以及目标Binder实体对象的成员变量has_strong_ref和has_weak_ref的值，来判断通知目标进程增加还是减少一个对应的Binder本地对象的强引用计数或者弱引用计数，判断规则如下所示。

（1）如果变量weak的值等于1，并且目标Binder实体对象的成员变量has_weak_ref的值等于0，那么就说明该Binder实体对象已经引用了一个Binder本地对象，但是还没有增加它的弱引用计数。因此，第32行到第36行代码就使用BR_INCREFS协议来请求增加对应的Binder本地对象的弱引用计数。

（2）如果变量strong的值等于1，并且目标Binder实体对象的成员变量has_strong_ref的值等于0，那么就说明该Binder实体对象已经引用了一个Binder本地对象，但是还没有增加它的强引用计数。因此，第38行到第42行代码就使用BR_ACQUIRE协议来请求增加对应的Binder本地对象的强引用计数。

（3）如果变量strong的值等于0，并且目标Binder实体对象的成员变量has_strong_ref的值等于1，那么就说明该Binder实体对象已经不再引用一个Binder本地对象了，但是还没有减少它的强引用计数。因此，第44行到第46行代码就使用BR_RELEASE协议来请求减少对应的Binder本地对象的强引用计数。

（4）如果变量weak的值等于0，并且目标Binder实体对象的成员变量has_weak_ref的值等于1，那么就说明该Binder实体对象已经不再引用一个Binder本地对象了，但是还没有减少它的弱引用计数。因此，第48行到第50行代码就使用BR_DECREFS协议来请求减少对应的Binder本地对象的弱引用计数。

第53行到第61行代码将前面准备好的协议以及协议内容写入到由Server进程所提供的一个用户空间缓冲区，然后返回到Server进程的用户空间。Server进程是通过Binder库提供的IPCThreadState接口来处理Binder驱动程序发送给它的协议的，具体来说，就是在IPCThreadState类的成员函数executeCommand中处理BR_INCREFS、BR_ACQUIRE、BR_DECREFS和BR_RELEASE这四个协议，如下所示。

frameworks/base/libs/binder/IPCThreadState.cpp

```
01 status_t IPCThreadState::executeCommand(int32_t cmd)
02 {
03     BBinder* obj;
04     RefBase::weakref_type* refs;
05     status_t result = NO_ERROR;
06
07     switch (cmd) {
08     ......
09     case BR_ACQUIRE:
10         refs = (RefBase::weakref_type*)mIn.readInt32();
11         obj = (BBinder*)mIn.readInt32();
12         ......
```

```
13          obj->incStrong(mProcess.get());
14          ......
15          mOut.writeInt32(BC_ACQUIRE_DONE);
16          mOut.writeInt32((int32_t)refs);
17          mOut.writeInt32((int32_t)obj);
18          break;
19      case BR_RELEASE:
20          refs = (RefBase::weakref_type*)mIn.readInt32();
21          obj = (BBinder*)mIn.readInt32();
22          ......
23          mPendingStrongDerefs.push(obj);
24          break;
25
26      case BR_INCREFS:
27          refs = (RefBase::weakref_type*)mIn.readInt32();
28          obj = (BBinder*)mIn.readInt32();
29          refs->incWeak(mProcess.get());
30          mOut.writeInt32(BC_INCREFS_DONE);
31          mOut.writeInt32((int32_t)refs);
32          mOut.writeInt32((int32_t)obj);
33          break;
34
35      case BR_DECREFS:
36          refs = (RefBase::weakref_type*)mIn.readInt32();
37          obj = (BBinder*)mIn.readInt32();
38          ......
39          mPendingWeakDerefs.push(refs);
40          break;
41      ......
42      }
43
44      ......
45      return result;
46  }
```

当Binder驱动程序请求Server进程增加或者减少一个Binder本地对象的强引用计数或者弱引用计数时，会将该Binder本地对象内部的一个弱引用计数对象（weakref_type），以及该Binder本地对象的地址传递过来，因此，IPCThreadState类的成员函数executeCommand就可以知道要修改的是哪一个Binder本地对象的引用计数了。Binder驱动程序传递给Server进程的数据保存在IPCThreadState类的成员变量mIn中，第10行和第11行、第20行和第21行、第27行和第28行，以及第36行和第37行就分别从它里面取出上述两个地址，并且分别将它转换为一个weakref_type对象refs和一个BBinder对象obj。

对于BR_ACQUIRE和BR_INCREFS协议，Server进程会马上增加目标Binder本地对象的强引用计数和弱引用计数，如第13行和第29行代码所示，并且使用BC_ACQUIRE_DONE和BC_INCREFS_DONE协议来通知Binder驱动程序，它已经增加相应的Binder本地对象的引用计数了。对于BR_RELEASE和BR_DECREFS协议，Server进程却不急于去处理，而是将它们缓存在IPCThreadState类的成员变量mPendingStrongDerefs和mPendingWeakDerefs中，如第23行和第39行代码所示，等到Server进程下次使用IO控制命令BINDER_WRITE_READ进入Binder驱动程序之前，再来处理它们。这是因为增加一个Binder本地对象的引用计数是一件紧急的事情，必须要马上处理；否则，该Binder本地对象就有可能提前被销毁。相反，减少一个Binder本地对象的引用计数是一件不重要的事情，至多就是延迟了Binder本地对象的生命周期。这样做的好处就是可以让Server进程优先去处理其他更重要的事情。

至此，我们就分析完成Binder本地对象的生命周期了。接下来，我们继续分析Binder实体对象的生命周期。

5.4.2　Binder实体对象的生命周期

Binder实体对象是一个类型为binder_node的对象，它是在Binder驱动程序中创建的，并且被Binder驱动程序中的Binder引用对象所引用。

当Client进程第一次引用一个Binder实体对象时，Binder驱动程序就会在内部为它创建一个Binder引用对象。例如，当Client进程通过Service Manager来获得一个Service组件的代理对象接口时，Binder驱动程序就会找到与该Service组件对应的Binder实体对象，接着再创建一个Binder引用对象来引用它。这时候就需要增加被引用的Binder实体对象的引用计数。相应地，当Client进程不再引用一个Service组件时，它也会请求Binder驱动程序释放之前为它所创建的一个Binder引用对象。这时候就需要减少该Binder引用对象所引用的Binder实体对象的引用计数。

在Binder驱动程序中，增加一个Binder实体对象的引用计数是通过函数binder_inc_node来实现的，如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static int
02 binder_inc_node(struct binder_node *node, int strong, int internal,
03                  struct list_head *target_list)
04 {
05         if (strong) {
06                 if (internal) {
07                         if (target_list == NULL &&
08                             node->internal_strong_refs == 0 &&
09                             !(node == binder_context_mgr_node &&
10                             node->has_strong_ref)) {
11                                 printk(KERN_ERR "binder: invalid inc strong "
12                                         "node for %d\n", node->debug_id);
13                                 return -EINVAL;
14                         }
15                         node->internal_strong_refs++;
16                 } else
17                         node->local_strong_refs++;
18                 if (!node->has_strong_ref && target_list) {
19                         list_del_init(&node->work.entry);
20                         list_add_tail(&node->work.entry, target_list);
21                 }
22         } else {
23                 if (!internal)
24                         node->local_weak_refs++;
25                 if (!node->has_weak_ref && list_empty(&node->work.entry)) {
26                         if (target_list == NULL) {
27                                 printk(KERN_ERR "binder: invalid inc weak node "
28                                         "for %d\n", node->debug_id);
29                                 return -EINVAL;
30                         }
31                         list_add_tail(&node->work.entry, target_list);
32                 }
33         }
34         return 0;
35 }
```

第一个参数node表示要增加引用计数的Binder实体对象；第二个参数strong表示要增加强引用计数还是弱引用计数；第三个参数internal用来区分增加的是内部引用计数还是外部引用计数；第四个参数target_list指向一个目标进程或者目标线程的todo队列，当它不为NULL时，就表示增加了Binder实体对象node的引用计数之后，要相应地增加它所引用的Binder本地对象的引用计数。

在前面的5.1.1小节中介绍结构体binder_node时提到，每一个Binder实体对象都有一个Binder引用对象列表refs，以及三个引用计数internal_strong_refs、local_strong_refs和local_weak_refs。其中，local_strong_refs和local_weak_refs分别表示该Binder实体对象的内部强引用计数和弱引用计数；而internal_strong_refs是一个外部强引用计数，用来描述有多少个Binder引用对象是通过强引用计数来引用该Binder实体对象的。内部引用计数是相对于该Binder实体对象所在的Server进程而言的，而外部引用计数是相对于引用了该Binder实体对象的Client进程而言的。当Binder驱动程序请求Server进程中的某一个Binder本地对象来执行一个操作时，它就会增加与该Binder本地对象对应的Binder实体对象的内部引用计数，避免该Binder实体对象被过早地销毁；而当一个Client进程通过一个Binder引用对象来引用一个Binder实体对象时，Binder驱动程序就会增加它的外部引用计数，也是避免该Binder实体对象被过早地销毁。读者可能会觉得奇怪，为什么一个Binder实体对象的外部引用计数只有强引用计数internal_strong_refs，而没有一个对应的弱引用计数internal_weak_refs呢？原来，Binder实体对象的Binder引用对象列表refs的大小就已经隐含了它的外部弱引用计数，因此，就不需要使用一个额外的弱引用计数internal_weak_refs来描述它的外部弱引用计数了。

了解了这些基础知识之后，函数binder_inc_node的实现原理就容易理解了。

如果参数strong和internal的值均等于1，即第5行和第6行的if语句均为true，那么就表示要增加Binder实体对象node的外部强引用计数internal_strong_refs。当参数internal的值等于1时，意味着此时是Binder驱动程序在为一个Client进程增加Binder实体对象node的外部强引用计数。这时候如果它的外部强引用计数internal_strong_refs等于0，并且它还没有增加对应的Binder本地对象的强引用计数，即它的成员变量has_strong_ref也等于0，那么就需要增加该Binder本地对象的强引用计数了；否则，该Binder本地对象就可能会过早地被销毁。在这种情况下，就必须指定参数target_list的值，而且它必须指向该Binder本地对象所在进程的todo队列，以便函数binder_inc_node可以往里面加入一个工作项来通知该进程增加对应的Binder本地对象的强引用计数，如第18行到第21行的if语句所示。但是有一种特殊情况，即Binder实体对象node引用的Binder本地对象是Service Manager时，可以不指定参数target_list的值，因为这个Binder本地对象是不会被销毁的。在后面的5.6小节中，我们再详细分析Service Manager的实现原理。

如果参数strong和internal的值分别等于1和0，即第5行的if语句和第6行的if语句分别为true和false，那么函数binder_inc_node就只是简单地增加Binder实体对象node的内部强引用计数local_strong_refs，如第17行代码所示。接着同样调用第18行到第21行的if语句来判断是否需要增加相应的Binder本地对象的强引用计数。

如果参数strong和internal的值均等于0，即第5行的if语句和第23行的if语句分别为false和true，那么函数binder_inc_node就先增加Binder实体对象node的内部弱引用计数local_weak_refs。接着第25行判断Binder实体对象node的成员变量has_weak_ref是否等于0。如果等于0，就说明Binder驱动程序还没有增加过与Binder实体对象node对应的Binder本地对象的弱引用计数，因此，这时候参数target_list就必须指向该Binder本地对象所在进程的todo队列，以便函数binder_inc_node可以往里面加入一个工作项来通知该进程增加对应的Binder本地对象的弱引用计数，如第31行代码所示。

如果参数strong和internal的值分别等于0和1，那么Binder驱动程序在调用函数binder_inc_node之前，就已经将一个对应的Binder引用对象添加到Binder实体对象node的Binder引用对象列表中，即相当于增加了Binder实体对象node的外部弱引用计数，因此，这时候函数binder_inc_node就没有增加相应计数的操作。不过，它仍然需要调用第25行的if语句来判断是否要增加相应的Binder本地对象的弱引用计数。

在Binder驱动程序中，减少一个Binder实体对象的引用计数是通过函数binder_dec_node来实现的，如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static int
02 binder_dec_node(struct binder_node *node, int strong, int internal)
03 {
04         if (strong) {
05                 if (internal)
06                         node->internal_strong_refs--;
07                 else
08                         node->local_strong_refs--;
09                 if (node->local_strong_refs || node->internal_strong_refs)
10                         return 0;
11         } else {
12                 if (!internal)
13                         node->local_weak_refs--;
14                 if (node->local_weak_refs || !hlist_empty(&node->refs))
15                         return 0;
16         }
17         if (node->proc && (node->has_strong_ref || node->has_weak_ref)) {
18                 if (list_empty(&node->work.entry)) {
19                         list_add_tail(&node->work.entry, &node->proc->todo);
20                         wake_up_interruptible(&node->proc->wait);
21                 }
22         } else {
23                 if (hlist_empty(&node->refs) && !node->local_strong_refs &&
24                     !node->local_weak_refs) {
25                         list_del_init(&node->work.entry);
26                         if (node->proc) {
27                                 rb_erase(&node->rb_node, &node->proc->nodes);
28                                 ......
29                         } else {
30                                 hlist_del(&node->dead_node);
31                                 .....
32                         }
33                         kfree(node);
34                         ......
35                 }
36         }
37
38         return 0;
39 }
```

第一个参数node表示要减少引用计数的Binder实体对象；第二个参数strong表示要减少强引用计数还是弱引用计数；第三个参数internal用来区分减少的是内部引用计数，还是外部引用计数。

如果参数strong和internal的值均等于1，即第4行和第5行的if语句均为true，那么就表示要减少Binder实体对象node的外部强引用计数internal_strong_refs。

如果参数strong和internal的值分别等于1和0，即第4行和第5行的if语句分别为true和false，那么就表示要减少Binder实体对象node的内部强引用计数local_strong_refs。

如果参数strong和internal的值均等于0，即第4行和第12行的if语句分别为false和true，那么就表示要减少Binder实体对象node的内部弱引用计数local_weak_refs。

如果参数strong和internal的值分别等于0和1，那么Binder驱动程序在调用函数binder_dec_node之前，就已经将一个对应的Binder引用对象从Binder实体对象node的Binder引用对象列表删除了，即相当于减少了Binder实体对象node的外部弱引用计数，因此，这时候函数binder_dec_node就没有减少相应计数的操作。

函数binder_dec_node减少了Binder实体对象node的强引用计数或者弱引用计数之后，如果它的强引用计数或者弱引用计数不等于0，即第9行或者第14行的if语句为true，那么函数binder_dec_node就不用往下处理了；否则，就需要减少与Binder实体对象node对应的Binder本地对象的引用计数。

如果函数执行到第17行，就说明Binder实体对象node的强引用计数或者弱引用计数等于0了。这时候如果它的成员变量has_strong_ref和has_weak_ref的其中一个等于1，那么就需要减少与Binder实体对象node对应的Binder本地对象的强引用计数或者弱引用计数，因为此时Binder实体对象node不需要通过强引用计数或者弱引用计数来引用相应的Binder本地对象，但是之前增加过它的强引用计数或者弱引用计数。第18行判断是否已经将一个类型为BINDER_WORK_NODE的工作项添加到要减少引用计数的Binder本地对象所在进程的todo队列中。如果不是，那么第19行就添加该工作项，并且第20行调用函数wake_up_interruptible唤醒该进程来处理该工作项，以便它可以马上执行减少相应的Binder本地对象的引用计数的工作。

如果第17行的if语句为false，那么要么是Binder实体对象node的宿主进程结构体为NULL，要么是它的成员变量has_strong_ref和has_weak_ref都等于0。这时候如果Binder实体对象node的所有引用计数均等于0，即第23行和第24行的if语句为true，那么就需要销毁该Binder实体对象node。在销毁Binder实体对象node之前，如果发现它的宿主进程结构体为NULL，即第26行的if语句为false，就说明该Binder实体对象node之前是保存在一个死亡Binder实体对象列表中的，因此，第30行就将它从该列表中删除。如果它的宿主进程结构体不为NULL，即第26行的if语句为true，就说明它是保存在其宿主进程结构体的Binder实体对象红黑树中的，因此，第27行就将它从该红黑树中删除。最后，第33行调用函数kfree来销毁Binder实体对象node。

至此，我们就分析完成Binder实体对象的生命周期了。从分析的过程中，可以看到一个有趣的事实，如图5-18所示。

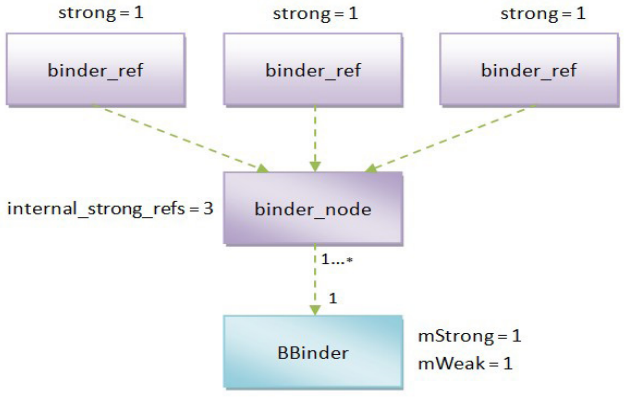


图5-18　Binder实体对象与Binder本地对象的引用关系

假设有三个Binder引用对象引用了一个Binder实体对象，并且这三个Binder引用对象的强引用计数都等于1。每当一个Binder引用对象的强引用计数由0变为1时，它都会增加对应的Binder实体体象的外部强引用计数internal_strong_refs，因此，图5-18中的Binder实体对象的外部强引用计数internal_strong_refs等于3。但是该Binder实体对象只增加了它所对应的Binder本地对象的1个强引用计数和1个弱引用计数，即一个Binder实体对象的外部强引用计数internal_strong_refs与它所对应的Binder本地对象的引用计数是多对一的关系。这样做的好处是可以减少Binder驱动程序和Server进程之间的交互，即可以减少它们之间的BR_INCREFS、BR_ACQUIRE、BR_DECREFS和BR_RELEASE协议来往。

接下来，我们继续分析Binder引用对象的生命周期。

5.4.3　Binder引用对象的生命周期

Binder引用对象是一个类型为binder_ref的对象，它是在Binder驱动程序中创建的，并且被用户空间中的Binder代理对象所引用。

当Client进程引用了Server进程中的一个Binder本地对象时，Binder驱动程序就会在内部为它创建一个Binder引用对象。由于Binde引用对象是运行在内核空间的，而引用了它的Binder代理对象是运行在用户空间的，因此，Client进程和Binder驱动程序就需要约定一套规则来维护Binder引用对象的引用计数，避免它们在还被Binder代理对象引用的情况下被销毁。

这套规则可以划分为BC_ACQUIRE、BC_INCREFS、BC_RELEASE和BC_DECREFS四个协议，分别用来增加和减少一个Binder引用对象的强引用计数和弱引用计数。相关的代码实现在Binder驱动程序的函数binder_thread_write中，如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 int
02 binder_thread_write(struct binder_proc *proc, struct binder_thread *thread,
03                     void __user *buffer, int size, signed long *consumed)
04 {
05         uint32_t cmd;
06         void __user *ptr = buffer + *consumed;
07         void __user *end = buffer + size;
08
09         while (ptr < end && thread->return_error == BR_OK) {
10                 if (get_user(cmd, (uint32_t __user *)ptr))
11                         return -EFAULT;
12                 ptr += sizeof(uint32_t);
13                 ......
14                 switch (cmd) {
15                 case BC_INCREFS:
16                 case BC_ACQUIRE:
17                 case BC_RELEASE:
18                 case BC_DECREFS: {
19                         uint32_t target;
20                         struct binder_ref *ref;
21                         const char *debug_string;
22
23                         if (get_user(target, (uint32_t __user *)ptr))
24                                 return -EFAULT;
25                         ptr += sizeof(uint32_t);
26                         if (target == 0 && binder_context_mgr_node &&
27                             (cmd == BC_INCREFS || cmd == BC_ACQUIRE)) {
28                                 ref = binder_get_ref_for_node(proc,
29                                                 binder_context_mgr_node);
30                                 ......
31                         } else
32                                 ref = binder_get_ref(proc, target);
33                         ......
34                         switch (cmd) {
35                         case BC_INCREFS:
36                                 debug_string = "IncRefs";
37                                 binder_inc_ref(ref, 0, NULL);
38                                 break;
39                         case BC_ACQUIRE:
40                                 debug_string = "Acquire";
41                                 binder_inc_ref(ref, 1, NULL);
42                                 break;
43                         case BC_RELEASE:
44                                 debug_string = "Release";
45                                 binder_dec_ref(ref, 1);
46                                 break;
47                         case BC_DECREFS:
48                         default:
49                                 debug_string = "DecRefs";
50                                 binder_dec_ref(ref, 0);
51                                 break;
```

```
52                          }
53                          ......
54                          break;
55                  }
56                  ......
57                  }
58
59                  ......
60                  *consumed = ptr - buffer;
61          }
62          return 0;
63  }
```

Binder驱动程序会根据Client进程传进来的句柄值找到要修改引用计数的Binder引用对象。这个句柄值是Binder驱动程序为Client进程创建的，用来关联用户空间的Binder代理对象和内核空间的Binder引用对象。在Binder驱动程序中，有一个特殊的句柄值，它的值等于0，用来描述一个与Service Manager关联的Binder引用对象。

第23行首先获得由Client进程传进来的句柄值，并且保存在变量target中。接着第26行判断该句柄值是否等于0。如果等于0，那么第26行就会继续判断Binder驱动程序中是否存在一个引用了Service Manager的Binder实体对象，即全局变量binder_context_mgr_node的值是否不等于NULL。如果是，并且要执行的操作是增加它的强引用计数或者弱引用计数，那么第28行和第29行就会调用函数binder_get_ref_for_node来获得一个与Service Manager对应的Binder引用对象。如果Binder驱动程序之前尚未为Client进程创建过这个Binder引用对象，那么函数binder_get_ref_for_node就会为它创建一个，以便Client进程可以引用它。如果Client进程传进来的句柄值不等于0，那么第32行就会调用函数binder_get_ref来获取对应的Binder引用对象。如果对应的Binder引用对象不存在，那么函数binder_get_ref不会根据这个句柄值来创建一个Binder引用对象返回给调用者。因为句柄值本来就是在创建Binder引用对象的时候分配的，即先有Binder引用对象，再有句柄值，只不过值等于0的句柄比较特殊，它允许先有句柄值，再有Binder引用对象。

得到了目标Binder引用对象ref之后，第34行到第52行的switch语句根据不同的协议来增加或者减少它的强引用计数或者弱引用计数。其中，BC_ACQUIRE和BC_INCREFS协议用来增加一个Binder引用对象的强引用计数和弱引用计数，调用的函数为binder_inc_ref；而BC_RELEASE和BC_DECREFS协议用来减少一个Binder引用对象的强引用计数和弱引用计数，调用的函数为binder_dec_ref。下面我们就分别分析函数binder_inc_ref和binder_dec_ref的实现。

函数binder_inc_ref的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static int
02 binder_inc_ref(
03         struct binder_ref *ref, int strong, struct list_head *target_list)
04 {
05         int ret;
06         if (strong) {
07                 if (ref->strong == 0) {
08                         ret = binder_inc_node(ref->node, 1, 1, target_list);
09                         if (ret)
10                                 return ret;
11                 }
12                 ref->strong++;
13         } else {
14                 if (ref->weak == 0) {
15                         ret = binder_inc_node(ref->node, 0, 1, target_list);
16                         if (ret)
17                                 return ret;
```

```
18                 }
19                 ref->weak++;
20         }
21         return 0;
22 }
```

第一个参数ref用来描述要操作的Binder引用对象；第二个参数strong用来描述是要增加Binder引用对象ref的强引用计数还是弱引用计数；第三个参数target_list指向一个目标进程或者目标线程的todo队列，当它不为NULL时，就表示增加了Binder引用对象ref的引用计数之后，要相应地增加与它所引用的Binder实体对象对应的Binder本地对象的引用计数。

如果参数strong的值等于1，即第6行的if语句为true，那么就说明要增加Binder引用对象ref的强引用计数，因此，第12行就增加它的成员变量strong的值。如果一个Binder引用对象的强引用计数由0变成1，那么第8行还需要调用函数binder_inc_node来增加它所引用的Binder实体对象的外部强引用计数internal_strong_refs，避免它在还被一个Binder引用对象引用的情况下被销毁。

如果参数strong的值等于0，即第6行的if语句为false，那么就说明要增加Binder引用对象ref的弱引用计数，因此，第19行就增加它的成员变量weak的值。同样，如果一个Binder引用对象的弱引用计数由0变成1，那么第15行还需要调用函数binder_inc_node来增加它所引用的Binder实体对象的外部弱引用计数，避免它在还被一个Binder引用对象引用的情况下被销毁。

函数binder_inc_node的实现在前面的5.4.2小节已经分析过了。接下来，我们继续分析函数binder_dec_ref的实现，如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static int
02 binder_dec_ref(struct binder_ref *ref, int strong)
03 {
04         if (strong) {
05                 if (ref->strong == 0) {
06                         binder_user_error("binder: %d invalid dec strong, "
07                                           "ref %d desc %d s %d w %d\n",
08                                           ref->proc->pid, ref->debug_id,
09                                           ref->desc, ref->strong, ref->weak);
10                         return -EINVAL;
11                 }
12                 ref->strong--;
13                 if (ref->strong == 0) {
14                         int ret;
15                         ret = binder_dec_node(ref->node, strong, 1);
16                         if (ret)
17                                 return ret;
18                 }
19         } else {
20                 if (ref->weak == 0) {
21                         binder_user_error("binder: %d invalid dec weak, "
22                                           "ref %d desc %d s %d w %d\n",
23                                           ref->proc->pid, ref->debug_id,
24                                           ref->desc, ref->strong, ref->weak);
25                         return -EINVAL;
26                 }
27                 ref->weak--;
28         }
29         if (ref->strong == 0 && ref->weak == 0)
30                 binder_delete_ref(ref);
31         return 0;
32 }
```

第一个参数ref用来描述要操作的Binder引用对象；第二个参数strong用来描述是要减少Binder引用对象ref的强引用计数还是弱引用计数。

如果参数strong的值等于1，即第4行的if语句为true，就说明要减少Binder引用对象ref的强引用计数，因此，第12行就减少它的成员变量strong的值。在减少一个Binder引用对象的强引用计数之前，首先要判断它的强用计数是否大于0。如果不是，即第5行的if语句为true，那么就说明出现了不对称的强引用计数增加和减少操作。如果一个Binder引用对象的强引用计数由1变成0，那么第15行就需要调用函数binder_dec_node来减少它所引用的Binder实体对象的外部强引用计数internal_strong_refs，表示它不再被该Binder引用对象通过外部强引用计数来引用了。

如果参数strong的值等于0，即第4行的if语句为false，就说明要减少Binder引用对象的弱引用计数，因此，第27行就减少它的成员变量weak的值。同样，在减少一个Binder引用对象的弱引用计数之前，首先要判断它的弱引用计数是否大于0。如果不是，即第20行的if语句为true，就说明出现了不对称的弱引用计数增加和减少操作。

一个Binder引用对象的生命周期同时受到它的强引用计数和弱引用计数影响，因此，当它们都等于0时，即第29行的if语句为true时，第30行就需要调用函数binder_delete_ref来销毁该binder_ref对象了。

函数binder_delete_ref的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static void
02 binder_delete_ref(struct binder_ref *ref)
03 {
04         ......
05         rb_erase(&ref->rb_node_desc, &ref->proc->refs_by_desc);
06         rb_erase(&ref->rb_node_node, &ref->proc->refs_by_node);
07         if (ref->strong)
08                 binder_dec_node(ref->node, 1, 1);
09         hlist_del(&ref->node_entry);
10         binder_dec_node(ref->node, 0, 1);
11         if (ref->death) {
12                 ......
13                 list_del(&ref->death->work.entry);
14                 kfree(ref->death);
15                 ......
16         }
17         kfree(ref);
18         ......
19 }
```

在前面的5.1.1小节中介绍结构体binder_proc时提到，一个Binder引用对象分别保存在其宿主进程的两个红黑树中，因此，第5行和第6行就将它从这两个红黑树中移除。同时，在介绍结构体binder_node时提到，一个Binder引用对象还保存在它所引用的Binder实体对象的Binder引用对象列表中，因此，第9行就将它从该列表中移除。

如果一个Binder引用对象是被强行销毁的，即在它的强引用计数还大于0的情况下销毁，那么第8行就需要调用函数binder_dec_node来减少它所引用的Binder实体对象的外部强引用计数internal_strong_refs，因为当该Binder引用对象销毁之后，它所引用的Binder实体对象就少了一个Binder引用对象通过外部强引用计数来引用了。

第10行调用函数binder_dec_node来减少一个即将要销毁的Binder引用对象所引用的Binder实体对象的外部弱引用计数。但是从前面5.4.2小节的内容可以知道，一个Binder实体对象是没有一个类似于internal_strong_refs的外部弱引用计数的，这里调用函数binder_dec_node只是为了检查是否需要减少相应的Binder本地对象的弱引用计数。

第11行检查即将要销毁的Binder引用对象是否注册有一个死亡接收通知。如果是，那么第13行和第14行就删除以及销毁该死亡接收通知。在接下来的5.4.4小节中，我们再详细分析Binder驱动程序中

的Service组件死亡通知机制。

最后，第17行就调用函数kfree来销毁Binder引用对象ref了。

至此，我们就分析完成Binder引用对象的生命周期了。从分析的过程中，可以看到一个有趣的事实，如图5-19所示。

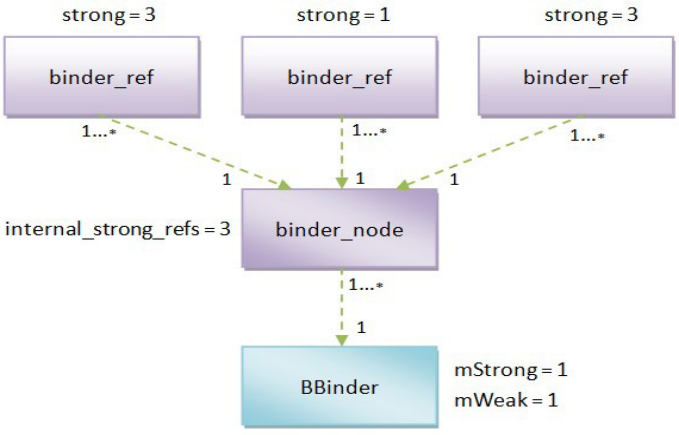


图5-19　Binder引用对象与Binder实体对象的引用关系

假设有三个Binder引用对象引用了一个Binder实体对象，其中，有两个Binder引用对象的强引用计数等于3，另外一个Binder引用对象的强引用计数等于1。每当一个Binder引用对象的强引用计数由0变为1时，它都会增加对应的Binder实体对象的外部强引用计数internal_strong_refs。因此，图5-19中的Binder实体对象的外部强引用计数internal_strong_refs等于3，小于所有引用了它的Binder引用对象的强引用计数之和，即一个Binder引用对象的强引用计数与它所引用的Binder实体对象的外部强引用计数是多对一的关系。这样做的好处是可以减少执行修改Binder实体对象的引用计数的操作。

接下来，我们继续分析Binder代理对象的生命周期。

5.4.4　Binder代理对象的生命周期

Binder代理对象是一个类型为BpBinder的对象，它是在用户空间中创建的，并且运行在Client进程中。与Binder本地对象类似，Binder代理对象一方面会被运行在Client进程中的其他对象引用，另一方面它也会引用Binder驱动程序中的Binder引用对象。由于BpBinder类继承了RefBase类，因此，Client进程中的其他对象可以简单地通过智能指针来引用这些Binder代理对象，以便可以控制它们的生命周期。由于Binder驱动程序中的Binder引用对象是运行在内核空间的，Binder代理对象就不能通过智能指针来引用它们，因此，Client进程就需要通过BC_ACQUIRE、BC_INCREFS、BC_RELEASE和BC_DECREFS四个协议来引用Binder驱动程序中的Binder引用对象，如前面5.4.3小节的内容所示。

前面提到，每一个Binder代理对象都是通过一个句柄值来和一个Binder引用对象关联的，而Client进程就是通过这个句柄值来维护运行在它里面的Binder代理对象的。具体来说，就是Client进程会在内部建立一个handle_entry类型的Binder代理对象列表，它以句柄值作为关键字来维护它内部所有的Binder代理对象。

结构体handle_entry内部有一个IBinder指针，指向一个BpBinder对象，即一个Binder代理对象。每当Client进程接收到从Binder驱动程序传过来的句柄值时，它就会以该句柄值作为关键字在其内部的Binder代理对象列表中检查是否已经存在一个对应的handle_entry结构体。如果存在，那么Client进程就会使用该handle_entry结构体内部的IBinder指针所指向的BpBinder对象来和Server进程进行通信；否则，Client进程就会先创建一个BpBinder对象，并且将它保存在一个handle_entry结构体中，以及将该

handle_entry结构体保存在内部的Binder代理对象列表中。这样做的好处是可以避免创建重复的Binder代理对象来引用Binder驱动程序中的同一个Binder引用对象，因为每一个Binder代理对象在创建时都会去增加相应的Binder引用对象的引用计数。

Binder代理对象的创建过程如下所示。

frameworks/base/libs/binder/BpBinder.cpp

```
1 BpBinder::BpBinder(int32_t handle)
2     : mHandle(handle)
3     ......
4 {
5     ......
6
7     extendObjectLifetime(OBJECT_LIFETIME_WEAK);
8     IPCThreadState::self()->incWeakHandle(handle);
9 }
```

第7行设置Binder代理对象的生命周期受到弱引用计数影响，接着第8行调用当前线程内部的IPCThreadState对象的成员函数incWeakHandle来增加相应的Binder引用对象的弱引用计数。

frameworks/base/libs/binder/IPCThreadState.cpp

```
1 void IPCThreadState::incWeakHandle(int32_t handle)
2 {
3     ......
4     mOut.writeInt32(BC_INCREFS);
5     mOut.writeInt32(handle);
6 }
```

IPCThreadState类的成员函数incWeakHandle将增加Binder引用对象的弱引用计数的操作缓存在内部的一个成员变量mOut中，等到下次使用IO控制命令BINDER_WRITE_READ进入到Binder驱动程序时，再请求Binder驱动程序增加相应的Binder引用对象的弱引用计数。

另外，当一个Binder代理对象销毁时，当前线程就会调用其内部的IPCThreadState对象的成员函数decWeakHandle来减少相应的Binder引用对象的弱引用计数。

frameworks/base/libs/binder/BpBinder.cpp

```
01 BpBinder::~BpBinder()
02 {
03     ......
04
05     IPCThreadState* ipc = IPCThreadState::self();
06
07     ......
08
09     if (ipc) {
10         ......
11         ipc->decWeakHandle(mHandle);
12     }
13 }
```

IPCThreadState类的成员函数decWeakHandle的实现如下所示。

frameworks/base/libs/binder/IPCThreadState.cpp

```
1 void IPCThreadState::decWeakHandle(int32_t handle)
2 {
3     ......
4     mOut.writeInt32(BC_DECREFS);
5     mOut.writeInt32(handle);
6 }
```

IPCThreadState类的成员函数decWeakHandle同样是将减少Binder引用对象的弱引用计数的操作缓存在内部的一个成员变量mOut中，等到下次使用IO控制命令BINDER_WRITE_READ进入到Binder驱动

程序时，再请求Binder驱动程序减少相应的Binder引用对象的弱引用计数。

前面提到，一个Binder代理对象可能会被用户空间的其他对象引用，当其他对象通过强指针来引用这些Binder代理对象时，Client进程就需要请求Binder驱动程序增加相应的Binder引用对象的强引用计数。为了减少Client进程和Binder驱动程序的交互开销，当一个Binder代理对象第一次被强指针引用时，Client进程才会请求Binder驱动程序增加相应的Binder引用对象的强引用计数；同时，当一个Binder代理对象不再被任何强指针引用时，Client进程就会请求Binder驱动程序减少相应的Binder引用对象的强引用计数。

在前面第3章介绍Android系统的智能指针时提到，当一个对象第一次被强指针引用时，它的成员函数onFirstRef就会被调用；而当一个对象不再被任何强指针引用时，它的成员函数onLastStrongRef就会被调用。因此，接下来我们就分析Binder代理对象类BpBinder的成员函数onFirstRef和onLastStrongRef的实现。

BpBinder类的成员函数onFirstRef的实现如下所示。

frameworks/base/libs/binder/BpBinder.cpp

```
1 void BpBinder::onFirstRef()
2 {
3     ......
4     IPCThreadState* ipc = IPCThreadState::self();
5     if (ipc) ipc->incStrongHandle(mHandle);
6 }
```

它调用当前线程内部的IPCThreadState对象的成员函数incStrongHandle来增加相应的Binder引用对象的强引用计数。

frameworks/base/libs/binder/IPCThreadState.cpp

```
1 void IPCThreadState::incStrongHandle(int32_t handle)
2 {
3     ......
4     mOut.writeInt32(BC_ACQUIRE);
5     mOut.writeInt32(handle);
6 }
```

IPCThreadState类的成员函数incStrongHandle将增加Binder引用对象的强引用计数的操作缓存在内部的一个成员变量mOut中，等到下次使用IO控制命令BINDER_WRITE_READ进入到Binder驱动程序时，再请求Binder驱动程序增加相应的Binder引用对象的强引用计数。

BpBinder类的成员函数onLastStrongRef的实现如下所示。

frameworks/base/libs/binder/BpBinder.cpp

```
1 void BpBinder::onLastStrongRef(const void* id)
2 {
3     ......
4     IPCThreadState* ipc = IPCThreadState::self();
5     if (ipc) ipc->decStrongHandle(mHandle);
6 }
```

它调用当前线程内部的IPCThreadState对象的成员函数decStrongHandle来减少相应的Binder引用对象的强引用计数。

frameworks/base/libs/binder/IPCThreadState.cpp

```
1 void IPCThreadState::decStrongHandle(int32_t handle)
2 {
3     ......
4     mOut.writeInt32(BC_RELEASE);
5     mOut.writeInt32(handle);
6 }
```

IPCThreadState类的成员函数decStrongHandle同样是将减少Binder引用对象的强引用计数的操作缓存在内部的一个成员变量mOut中，等到下次使用IO控制命令BINDER_WRITE_READ进入到Binder驱动程序时，再请求Binder驱动程序减少相应的Binder引用对象的强引用计数。

至此，我们就分析完成Binder代理对象的生命周期了。从分析的过程中，可以看到一个有趣的事实，如图5-20所示。

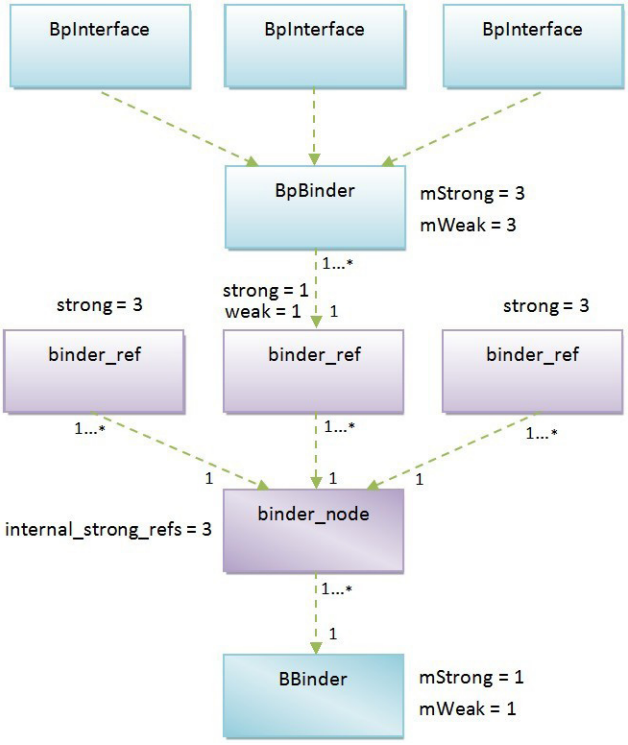


图5-20　Binder代理对象与Binder引用对象的引用关系

假设有Client进程将一个Binder代理对象封装成三个不同的BpInterface对象，并且这三个BpInterface对象都通过强指针来引用该Binder代理对象，因此，该Binder代理对象的强引用计数和弱引用计数就等于3。由于一个Binder代理对象只有在创建时，才会增加相应的Binder引用对象的弱引用计数，并且只有在第一次被强指针引用时才会增加相应的Binder引用对象的强引用计数，因此，图5-20中的Binder引用对象的强引用计数和弱引用计数均等于1，小于引用了它的Binder代理对象的强引用计数和弱引用计数，即一个Binder代理对象的引用计数与它所引用的Binder引用对象的引用计数是多对一的关系。这样做的好处是可以减少Client进程和Binder驱动程序之间的交互，即可以减少它们之间的BC_ACQUIRE、BC_INCREFS、BC_RELEASE和BC_DECREFS协议来往。

5.5　Binder对象死亡通知机制

在理想情况下，通过智能指针技术，以及BR_INCREFS、BR_ACQUIRE、BR_DECREFS、BR_RELEASE、BC_ACQUIRE、BC_INCREFS、BC_RELEASE和BC_DECREFS八个协议就可以正确地维护Binder进程间通信机制中各个Binder对象的生命周期。然而，在实际情况中，由于Client进程和Server进程在运行过程中可能会因为内部错误而异常退出，因此，运行在它们里面的Binder代理对象和Binder本地对象就会意外死亡，从而导致Binder进程间通信机制不能正确地维护它们与其他Binder对象的依赖关系。其中，最严重的是Binder本地对象的意外死亡，会导致依赖于它的Binder代理对象变得

无效。

在本节中，我们将分析Binder对象死亡通知机制，它可以监控到Binder本地对象的死亡事件，然后通知那些引用了它的Binder代理对象，从而在一定程度上解决无效Binder代理对象的问题。在这种死亡通知机制中，首先是Binder代理对象将一个死亡接收通知注册到Binder驱动程序中，然后当Binder驱动程序监控到它引用的Binder本地对象死亡时，Binder驱动程序就会向它发送一个死亡通知。另外，当一个Binder代理对象不需要接收它所引用的Binder本地对象的死亡通知时，它也可以注销之前所注册的死亡接收通知。

下面我们就首先分析Binder代理对象注册死亡接收通知的过程，然后分析Binder驱动程序发送Binder本地对象死亡接收通知的过程，最后分析Binder代理对象注销死亡接收通知的过程。

5.5.1　注册死亡接收通知

Binder代理对象在注册它所引用的Binder本地对象的死亡接收通知之前，首先要定义好死亡通知接收者。Binder库定义了死亡通知接收者必须要继承的基类DeathRecipient，它的实现如下所示。

frameworks/base/include/binder/IBinder.h

```
01 class IBinder : public virtual RefBase
02 {
03 public:
04     ......
05
06     class DeathRecipient : public virtual RefBase
07     {
08     public:
09         virtual void binderDied(const wp<IBinder>& who) = 0;
10     };
11
12     ......
13 };
```

自定义的死亡通知接收者必须要重写父类DeathRecipient的成员函数binderDied。当Binder驱动程序通知一个Binder代理对象它所引用的Binder本地对象已经死亡时，就会调用它所指定的死亡通知接收者的成员函数binderDied。

定义好死亡通知接收者之后，我们就可以调用Binder代理对象的成员函数linkToDeath来注册一个死亡接收通知了，它的实现如下所示。

frameworks/base/libs/binder/BpBinder.cpp

```
01 status_t BpBinder::linkToDeath(
02     const sp<DeathRecipient>& recipient, void* cookie, uint32_t flags)
03 {
04     Obituary ob;
05     ob.recipient = recipient;
06     ob.cookie = cookie;
07     ob.flags = flags;
08
09     ......
10
11     {
12         AutoMutex _l(mLock);
13
14         if (!mObitsSent) {
15             if (!mObituaries) {
16                 mObituaries = new Vector<Obituary>;
17                 ......
18
```

```
19                  IPCThreadState* self = IPCThreadState::self();
20                  self->requestDeathNotification(mHandle, this);
21                  self->flushCommands();
22              }
23              ssize_t res = mObituaries->add(ob);
24              return res >= (ssize_t)NO_ERROR ? (status_t)NO_ERROR : res;
25          }
26      }
27
28      return DEAD_OBJECT;
29  }
```

Binder代理对象的成员函数linkToDeath的另外两个参数cookie和flags也是用来标志一个死亡通知接收者的，在注销死亡接收通知时会用到。

我们可以为一个Binder代理对象同时注册多个死亡通知接收者，它们保存在Binder代理对象内部的一个讣告列表mObituaries中。每一个死亡通知接收者都使用一个Obituary对象来描述。Binder代理对象内部的成员变量mObitsSent用来表示Binder驱动程序是否已经向它发送过死亡通知，如果是，这个成员变量的值就等于1。在这种情况下，Binder代理对象的成员函数linkToDeath就会直接返回一个DEAD_OBJECT值，表示对应的Binder本地对象已经死亡了。如果Binder代理对象内部的成员变量mObitsSent的值等于0，那么它的成员函数linkToDeath首先将死亡通知接收者封装成一个Obituary对象，接着将它添加到内部的讣告列表mObituaries中。如果是第一次注册死亡通知接收者，那么函数就会调用当前线程中的IPCThreadState对象的成员函数requestDeathNotification来向Binder驱动程序注册一个死亡接收通知。

IPCThreadState类的成员函数requestDeathNotification的实现如下所示。

frameworks/base/libs/binder/IPCThreadState.cpp

```
1 status_t IPCThreadState::requestDeathNotification(int32_t handle, BpBinder* proxy)
2 {
3     mOut.writeInt32(BC_REQUEST_DEATH_NOTIFICATION);
4     mOut.writeInt32((int32_t)handle);
5     mOut.writeInt32((int32_t)proxy);
6     return NO_ERROR;
7 }
```

IPCThreadState类的成员函数requestDeathNotification将注册死亡接收通知的操作缓存在内部的一个成员变量mOut中，等到下次使用IO控制命令BINDER_WRITE_READ进入到Binder驱动程序时，再向Binder驱动程序注册死亡接收通知。从这里就可以看出，Client进程是通过BC_REQUEST_DEATH_NOTIFICATION协议来向Binder驱动程序注册死亡接收通知的。BC_REQUEST_DEATH_NOTIFICATION协议带有两个参数，分别是要注册死亡接收通知的Binder代理对象的句柄值和地址值。

回到Binder代理对象的成员函数linkToDeath中，接下来第21行就会调用当前线程的IPCThreadState对象的成员函数flushCommands促使当前线程马上通过IO控制命令BINDER_WRITE_READ进入到Binder驱动程序中，以便可以执行注册死亡接收通知的操作。

IPCThreadState类的成员函数flushCommands的实现如下所示。

frameworks/base/libs/binder/IPCThreadState.cpp

```
1 void IPCThreadState::flushCommands()
2 {
3     ......
4     talkWithDriver(false);
5 }
```

这里调用了IPCThreadState类的成员函数talkWithDriver来执行注册死亡通知的操作，主要就是通过IO控制命令BINDER_WRITE_READ来向Binder驱动程序发送一个BC_REQUEST_DEATH_NOTIFICATION协议。在后面的5.8小节中，我们再详细分析IPCThreadState类的成员函数talkWithDriver的实现。

进入到Binder驱动程序之后，函数binder_thread_write就会被调用来处理BC_REQUEST_DEATH_NOTIFICATION协议，如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 int
02 binder_thread_write(struct binder_proc *proc, struct binder_thread *thread,
03                void __user *buffer, int size, signed long *consumed)
04 {
05     uint32_t cmd;
06     void __user *ptr = buffer + *consumed;
07     void __user *end = buffer + size;
08
09     while (ptr < end && thread->return_error == BR_OK) {
10             if (get_user(cmd, (uint32_t __user *)ptr))
11                     return -EFAULT;
12             ptr += sizeof(uint32_t);
13             ......
14             switch (cmd) {
15             ......
16             case BC_REQUEST_DEATH_NOTIFICATION:
17             case BC_CLEAR_DEATH_NOTIFICATION: {
18                     uint32_t target;
19                     void __user *cookie;
20                     struct binder_ref *ref;
21                     struct binder_ref_death *death;
22
23                     if (get_user(target, (uint32_t __user *)ptr))
24                             return -EFAULT;
25                     ptr += sizeof(uint32_t);
26                     if (get_user(cookie, (void __user * __user *)ptr))
27                             return -EFAULT;
28                     ptr += sizeof(void *);
29                     ref = binder_get_ref(proc, target);
30                     ......
31
32                     if (cmd == BC_REQUEST_DEATH_NOTIFICATION) {
33                             if (ref->death) {
34                                     binder_user_error("binder: %d:%"
35                                             "d BC_REQUEST_DEATH_NOTI"
36                                             "FICATION death notific"
37                                             "ation already set\n",
38                                             proc->pid, thread->pid);
39                                     break;
40                             }
41                             death = kzalloc(sizeof(*death), GFP_KERNEL);
42                             ......
43                             INIT_LIST_HEAD(&death->work.entry);
44                             death->cookie = cookie;
45                             ref->death = death;
46                             if (ref->node->proc == NULL) {
47                                     ref->death->work.type = BINDER_WORK_DEAD_BINDER;
48                                     if (thread->looper & (BINDER_LOOPER_STATE_REGISTERED |
BINDER_LOOPER_STATE_ENTERED)) {
49                                             list_add_tail(&ref->death->work.entry, &thread->todo);
50                                     } else {
51                                             list_add_tail(&ref->death->work.entry, &proc->todo);
52                                             wake_up_interruptible(&proc->wait);
53                                     }
54                             }
55                     } else {
56                             ......
57                     }
58             } break;
```

```
59              ......
60              }
61              *consumed = ptr - buffer;
62      }
63      return 0;
64  }
```

第23行和第26行分别将Client进程传进来的Binder代理对象的句柄值和地址值取出来，并且保存在变量target和cookie中。第29行根据前面获得的句柄值target得到一个Binder引用对象ref。接下来，Binder驱动程序就要将一个死亡接收通知保存在该Binder引用对象ref的成员变量death中。

第33行检查之前是否已经为Binder引用对象ref注册过死亡接收通知。如果是，那么它的成员变量death就不等于NULL。Binder驱动程序是不允许重复地为同一个Binder引用对象注册死亡接收通知的，因此，在这种情况下，函数binder_thread_write就不会再继续往下处理了。如果是第一次为Binder引用对象ref注册死亡接收通知，那么第41行到第45行代码就首先创建一个binder_ref_death结构体，接着将用户空间传进来的参数cookie保存在它的成员变量cookie中，最后再将它保存在Binder引用对象ref的成员变量death中。

第45行代码执行完成之后，死亡接收通知的注册操作就完成了。但是可能会出现这样一种情况，即正在注册的Binder引用对象所引用的Binder本地对象已经死亡了，这时候Binder驱动程序就需要马上向Client进程发送一个死亡接收通知；否则，以后就没有机会发送这个通知了。现在的问题是，如何判断一个Binder引用对象所引用的Binder本地对象已经死亡了呢？一个Binder引用对象所引用的Binder实体对象保存在它的成员变量node中，而一个Binder实体对象的宿主进程结构体保存在它的成员变量proc中。如果一个Binder实体对象的宿主进程结构体为NULL，那么就说明它所引用的Binder本地对象已经死亡了，因为它所在的进程已经不存在了。在这种情况下，第47行到第53行代码就将一个类型为BINDER_WORK_DEAD_BINDER的工作项添加到当前线程或者当前线程所在的Client进程的todo队列中，以便可以向Client进程发送一个死亡接收通知。

第48行判断当前线程是否为Client进程的一个Binder线程。如果是，那么它的状态就应该为BINDER_LOOPER_STATE_REGISTERED 或者BINDER_LOOPER_STATE_ENTERED，这时候就可以将一个死亡接收通知就近发送给它处理了。如果当前线程不是一个Binder线程，那么Binder驱动程序就会将发送死亡接收通知的工作项添加到Client进程的todo队列中，等待其他的Binder线程来处理。

至此，一个死亡接收通知的注册过程就分析完成了。接下来，我们继续分析死亡接收通知的发送过程。

5.5.2 发送死亡接收通知

Server进程本来是应该常驻在系统中为Client进程提供服务的，但是可能会出现意外情况，导致它异常退出。Server进程一旦异常退出之后，运行在它里面的Binder本地对象就意外死亡了。这时候Binder驱动程序就应该向那些引用了它的Binder代理对象发送死亡接收通知，以便它们可以知道自己引用了一个无效的Binder本地对象。

现在的关键问题是，Binder驱动程序是如何知道一个Server进程退出运行了呢？Binder驱动程序将设备文件/dev/binder的释放操作方法设置为函数binder_release。Server进程在启动时，会调用函数open来打开设备文件/dev/binder。一方面，在正常情况下，它退出时会调用函数close来关闭设备文件/dev/binder，这时候就会触发函数binder_release被调用；另一方面，如果Server进程异常退出，即它没有正常关闭设备文件/dev/binder，那么内核就会负责关闭它，这时候也会触发函数binder_release被调用。因此，Binder驱动程序就可以在函数binder_release中检查进程退出时是否有Binder本地对象在里面运行。如果有，就说明它们是死亡了的Binder本地对象了。

Binder驱动程序中的函数binder_release的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static int binder_release(struct inode *nodp, struct file *filp)
02 {
03    struct binder_proc *proc = filp->private_data;
04    if (binder_proc_dir_entry_proc) {
05           char strbuf[11];
06           snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
07           remove_proc_entry(strbuf, binder_proc_dir_entry_proc);
08    }
09
10    binder_defer_work(proc, BINDER_DEFERRED_RELEASE);
11
12    return 0;
13 }
```

它除了检查进程中是否有Binder本地对象在运行之外，还会释放进程所占用的资源，例如，第4行到第8行的if语句块就是将之前为进程在/proc/binder/proc目录中创建的文件删除。由于检查进程中是否有Binder本地对象，以及资源释放操作都比较耗时，因此，第10行就调用函数binder_defer_work将一个BINDER_DEFERRED_RELEASE类型的延迟操作添加到一个全局的hash列表中。

BINDER_DEFERRED_RELEASE类型的延迟操作最终是由函数binder_deferred_release来执行的，它的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static void binder_deferred_release(struct binder_proc *proc)
02 {
03    struct hlist_node *pos;
04    ......
05
06    nodes = 0;
07    incoming_refs = 0;
08    while ((n = rb_first(&proc->nodes))) {
09           struct binder_node *node = rb_entry(n, struct binder_node, rb_node);
10
11           ......
12           if (hlist_empty(&node->refs)) {
13                  ......
14           } else {
15                  struct binder_ref *ref;
16                  ......
17
18                  hlist_for_each_entry(ref, pos, &node->refs, node_entry) {
19                         ......
20                         if (ref->death) {
21                                ......
22                                if (list_empty(&ref->death->work.entry)) {
23                                       ref->death->work.type = BINDER_WORK_DEAD_BINDER;
24                                       list_add_tail(&ref->death->work.entry, &ref->proc->todo);
25                                       wake_up_interruptible(&ref->proc->wait);
26                                } else
27                                       BUG();
28                         }
29                  }
30                  ......
31           }
32    }
33
34    ......
35 }
```

第8行到第32行的while循环检查目标进程proc的Binder实体对象列表nodes的每一个Binder实体对象。如果这些Binder实体对象的Binder引用对象列表refs不为空，那么第18行到第29行的for循环就检查这些Binder引用对象的成员变量death是否不等于NULL。如果是，就说明与这些Binder引用对象对应的Binder代理对象注册过死亡接收通知。因此，第23行和第24行就将一个BINDER_WORK_DEAD_BINDER类型的工作项添加到对应的Client进程的todo队列中，接着第25行调用函数wake_up_interruptible 来唤醒这些Client进程的Binder线程来处理这些死亡接收通知。

Client进程中的Binder线程在空闲时，会睡眠在Binder驱动程序的函数binder_thread_read中，因此，当它们被唤醒之后，就会继续执行函数binder_thread_read，并且检查自己以及宿主进程的todo队列，看看有没有工作项需要处理。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static int
02 binder_thread_read(struct binder_proc *proc, struct binder_thread *thread,
03     void  __user *buffer, int size, signed long *consumed, int non_block)
04 {
05     void __user *ptr = buffer + *consumed;
06     void __user *end = buffer + size;
07
08     ......
09
10     while (1) {
11             uint32_t cmd;
12             ......
13             struct binder_work *w;
14             ......
15
16             if (!list_empty(&thread->todo))
17                     w = list_first_entry(&thread->todo, struct binder_work, entry);
18             else if (!list_empty(&proc->todo) && wait_for_proc_work)
19                     w = list_first_entry(&proc->todo, struct binder_work, entry);
20             else {
21                     ......
22             }
23
24             ......
25
26             switch (w->type) {
27             ......
28             case BINDER_WORK_DEAD_BINDER:
29             case BINDER_WORK_DEAD_BINDER_AND_CLEAR:
30             case BINDER_WORK_CLEAR_DEATH_NOTIFICATION: {
31                     struct binder_ref_death *death = container_of(w, struct binder_ref_death, work);
32                     uint32_t cmd;
33                     if (w->type == BINDER_WORK_CLEAR_DEATH_NOTIFICATION)
34                             cmd = BR_CLEAR_DEATH_NOTIFICATION_DONE;
35                     else
36                             cmd = BR_DEAD_BINDER;
37                     if (put_user(cmd, (uint32_t __user *)ptr))
38                             return -EFAULT;
39                     ptr += sizeof(uint32_t);
40                     if (put_user(death->cookie, (void * __user *)ptr))
41                             return -EFAULT;
42                     ptr += sizeof(void *);
43                     ......
44
45                     if (w->type == BINDER_WORK_CLEAR_DEATH_NOTIFICATION) {
46                             ......
47                     } else
48                             list_move(&w->entry, &proc->delivered_death);
49                     if (cmd == BR_DEAD_BINDER)
```

```
50                              goto done; /* DEAD_BINDER notifications can cause transactions */
51              } break;
52              }
53
54              ......
55      }
56
57  done:
58
59      *consumed = ptr - buffer;
60      ......
61      return 0;
62  }
```

当前线程首先检查自己的todo队列有没有新工作项，如果没有，再检查宿主进程的todo队列有没有新工作项。由于前面Binder驱动程序将一个类型为BINDER_WORK_DEAD_BINDER的工作项添加到了Client进程的todo队列中，因此，它们最终会被Client进程中的Binder线程处理。

在前面5.1.1小节中介绍结构体binder_ref_death时提到，死亡接收通知的类型有三种，除了前面我们遇到的BINDER_WORK_DEAD_BINDER之外，还有BINDER_WORK_CLEAR_DEATH_NOTIFICATION和BINDER_WORK_DEAD_BINDER_AND_CLEAR两种。其中，BINDER_WORK_CLEAR_DEATH_NOTIFICATION类型的死亡接收通知其实只是用来告诉Client进程，它成功地注销了一个之前所注册的死亡接收通知；而BINDER_WORK_DEAD_BINDER_AND_CLEAR类型的死亡接收通知除了告诉Client进程，它已经成功地注销了一个死亡接收通知之外，还告诉Client进程，与该死亡接收通知所关联的一个Binder本地对象已经死亡了。

由于类型为BINDER_WORK_DEAD_BINDER、BINDER_WORK_CLEAR_DEATH_NOTIFICATION和BINDER_WORK_DEAD_BINDER_AND_CLEAR的工作项是嵌入在一个binder_ref_death结构体中的，因此，第31行就可以获得一个对应的binder_ref_death结构体，并且保存在变量death中。如果工作项的类型为BINDER_WORK_CLEAR_DEATH_NOTIFICATION，那么第34行就用协议BR_CLEAR_DEATH_NOTIFICATION_DONE来通知Client进程，它成功地注销了一个死亡接收通知；否则，第36行就使用协议BR_DEAD_BINDER来通知Client进程，它所引用的一个Binder本地对象已经死亡了，后者才是真正的死亡接收通知。

第37行和第40行分别将协议代码cmd和binder_ref_death结构体death的成员变量cookie写入到Client进程提供的一个用户空间缓冲区中。在前面5.5.1小节中介绍死亡接收通知的注册过程时提到，Client进程在注册一个死亡接收通知时，会将一个对应的Binder代理对象的地址值保存在一个binder_ref_death结构体的成员变量cookie中，因此，第40行写入到用户空间缓冲区的值实际上是用来通知Client进程，哪一个Binder代理对象所引用的Binder本地对象已经死亡了。

第45行的if语句检查当前正在处理的工作项的类型是否为BINDER_WORK_CLEAR_DEATH_NOTIFICATION。如果不是，那么第48就会将正在处理的工作项保存在Client进程的一个delivered_death队列中。等到Client进程处理完成这个工作项之后，即处理完成一个死亡接收通知之后，它就会使用协议BC_DEAD_BINDER_DOEN来通知Binder驱动程序，这时候Binder驱动程序就会将这个工作项从Client进程的delivered_death队列中移除。从这里就可以看出，如果一个进程的delivered_death队列不为空，那么就说明Binder驱动程序正在向它发送死亡接收通知。

当前线程返回到用户空间之后，就会在IPCThreadState类的成员函数executeCommand中处理协议BR_DEAD_BINDER，如下所示。

frameworks/base/libs/binder/IPCThreadState.cpp

```
01 status_t IPCThreadState::executeCommand(int32_t cmd)
02 {
03     BBinder* obj;
```

```
04      ......
05      status_t result = NO_ERROR;
06
07      switch (cmd) {
08      ......
09      case BR_DEAD_BINDER:
10          {
11              BpBinder *proxy = (BpBinder*)mIn.readInt32();
12              proxy->sendObituary();
13              mOut.writeInt32(BC_DEAD_BINDER_DONE);
14              mOut.writeInt32((int32_t)proxy);
15          } break;
16
17      ......
18      }
19
20      if (result != NO_ERROR) {
21          mLastError = result;
22      }
23
24      return result;
25  }
```

第11行首先获得用来接收死亡通知的Binder代理对象，接着第12行调用它的成员函数sendObituary来处理该死亡通知，最后第13行和第14行使用协议BC_DEAD_BINDER_DONE 来通知Binder驱动程序，它之前所发送的一个死亡接收通知已经处理完成了。

Binder代理对象的成员函数sendObituary的实现如下所示。

frameworks/base/libs/binder/BpBinder.cpp

```
01 void BpBinder::sendObituary()
02 {
03      ......
04
05      mLock.lock();
06      Vector<Obituary>* obits = mObituaries;
07      if(obits != NULL) {
08          ......
09          IPCThreadState* self = IPCThreadState::self();
10          self->clearDeathNotification(mHandle, this);
11          self->flushCommands();
12          mObituaries = NULL;
13      }
14      mObitsSent = 1;
15      mLock.unlock();
16
17      ......
18
19      if (obits != NULL) {
20          const size_t N = obits->size();
21          for (size_t i=0; i<N; i++) {
22              reportOneDeath(obits->itemAt(i));
23          }
24
25          delete obits;
26      }
27  }
```

第7行检查该Binder代理对象内部的讣告列表mObituaries是否为NULL。如果不是，就说明该Binder代理对象之前向Binder驱动程序注册过死亡接收通知。现在既然它所引用的Binder本地对象已经死亡了，第10行就调用当前线程中的IPCThreadState对象的成员函数clearDeathNotification来注销之前所注册的死亡接收通知，接着第11行通知该IPCThreadState对象立即执行这个注销操作。在接下来的

5.5.3小节中，我们再详细分析注销死亡接收通知的过程。

第14行将该Binder代理对象内部的成员变量mObitsSent的值设置为1，表示它已经接收过Binder驱动程序发送过来的死亡接收通知了。因此，如果以后再有其他组件调用它的成员函数linkToDeath来向它注册死亡通知接收者，那么它就会直接返回一个DEAD_OBJECT值给调用者，表示它们所关注的Binder本地对象已经死亡了。

第21行到第23行的for循环依次调用成员函数reportOneDeath来通知保存在讣告列表中的每一个死亡通知接收者，它们所关注的Binder本地对象已经死亡了。

Binder代理对象的成员函数reportOneDeath的实现如下所示。

frameworks/base/libs/binder/BpBinder.cpp

```
1 void BpBinder::reportOneDeath(const Obituary& obit)
2 {
3     sp<DeathRecipient> recipient = obit.recipient.promote();
4     ......
5     if (recipient == NULL) return;
6
7     recipient->binderDied(this);
8 }
```

注册到一个Binder代理对象中的死亡通知接收者被封装成一个Obituary对象，其中，指向死亡通知接收者的一个弱指针就保存在该Obituary对象的成员变量recipient中，因此，第3行首先将这个弱指针升级为一个强指针。如果得到的强指针不为NULL，那么就说明该死亡通知接收者还未被销毁，因此，第7行就调用它的成员函数binderDied来处理这个死亡接收通知。

至此，一个死亡接收通知的发送过程就分析完成了。接下来，我们继续分析死亡接收通知的注销过程。

5.5.3　注销死亡接收通知

当Client进程不再需要关注某一个Binder本地对象的死亡事件时，它就可以调用引用了该Binder本地对象的Binder代理对象的成员函数unlinkToDeath来注销前面所注册的一个死亡接收通知了。

Binder代理对象的成员函数unlinkToDeath的实现如下所示。

frameworks/base/libs/binder/BpBinder.cpp

```
01 status_t BpBinder::unlinkToDeath(
02     const wp<DeathRecipient>& recipient, void* cookie, uint32_t flags,
03     wp<DeathRecipient>* outRecipient)
04 {
05     AutoMutex _l(mLock);
06
07     if (mObitsSent) {
08         return DEAD_OBJECT;
09     }
10
11     const size_t N = mObituaries ? mObituaries->size() : 0;
12     for (size_t i=0; i<N; i++) {
13         const Obituary& obit = mObituaries->itemAt(i);
14         if ((obit.recipient == recipient
15                     || (recipient == NULL && obit.cookie == cookie))
16                 && obit.flags == flags) {
17             const uint32_t allFlags = obit.flags|flags;
18             if (outRecipient != NULL) {
19                 *outRecipient = mObituaries->itemAt(i).recipient;
20             }
21             mObituaries->removeAt(i);
22             if (mObituaries->size() == 0) {
```

```
23                 LOGV("Clearing death notification: %p handle %d\n", this, mHandle);
24                 IPCThreadState* self = IPCThreadState::self();
25                 self->clearDeathNotification(mHandle, this);
26                 self->flushCommands();
27                 delete mObituaries;
28                 mObituaries = NULL;
29             }
30             return NO_ERROR;
31         }
32     }
33
34     return NAME_NOT_FOUND;
35 }
```

第一个参数recipient表示前面注册到Binder代理对象内部的一个死亡通知接收者；第二个参数cookie和第三个参数flags是另外两个用来标志该死亡通知接收者的数据；第四个参数outRecipient是一个输出参数，用来返回前面所注册的一个死亡通知接收者。当第一个参数recipient为NULL时，第四个参数outRecipient才有用。

第7行检查Binder代理对象内部的成员变量mObitsSent的值是否等于1。如果等于1，就说明Binder驱动程序已经向该Binder代理对象发送过死亡接收通知了。在这种情况下，Binder代理对象就会主动向Binder驱动程序注销死亡接收通知，并且将它内部的死亡通知接收者列表清空。因此，这时候就不需要再执行注销死亡接收通知的操作了。

第12行到第32行的for循环依次检查Binder代理对象内部的讣告列表mObituaries是否存在Obituary对象与要注销的死亡通知接收者对应。如果存在，那么第21行就将它从讣告列表mObituaries中移除。第22行检查讣告列表mObituaries是否为空。如果是，那么第24行就会获得当前线程中的IPCThreadState对象，接着第25行调用这个IPCThreadState对象的成员函数clearDeathNotification来通知Binder驱动程序执行一个注销死亡接收通知的操作，最后第26行调用这个IPCThreadState对象的成员函数flushCommands触发该操作马上被执行。

IPCThreadState类的成员函数clearDeathNotification和flushCommands的实现如下所示。

frameworks/base/libs/binder/IPCThreadState.cpp

```
01 status_t IPCThreadState::clearDeathNotification(int32_t handle, BpBinder* proxy)
02 {
03     mOut.writeInt32(BC_CLEAR_DEATH_NOTIFICATION);
04     mOut.writeInt32((int32_t)handle);
05     mOut.writeInt32((int32_t)proxy);
06     return NO_ERROR;
07 }
08
09 void IPCThreadState::flushCommands()
10 {
11     ......
12     talkWithDriver(false);
13 }
```

它们使用了协议BC_CLEAR_DEATH_NOTIFICATION来请求Binder驱动程序执行一个注销死亡接收通知的操作。协议BC_CLEAR_DEATH_NOTIFICATION有两个参数，它们分别为要注销死亡接收通知的Binder代理对象的句柄值和地址值。

协议BC_CLEAR_DEATH_NOTIFICATION是在Binder驱动程序的函数binder_thread_write中处理的，它的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 int
02 binder_thread_write(struct binder_proc *proc, struct binder_thread *thread,
03                     void __user *buffer, int size, signed long *consumed)
```

```
04 {
05         uint32_t cmd;
06         void __user *ptr = buffer + *consumed;
07         void __user *end = buffer + size;
08
09         while (ptr < end && thread->return_error == BR_OK) {
10                 if (get_user(cmd, (uint32_t __user *)ptr))
11                         return -EFAULT;
12                 ptr += sizeof(uint32_t);
13                 ......
14                 switch (cmd) {
15                 ......
16                 case BC_REQUEST_DEATH_NOTIFICATION:
17                 case BC_CLEAR_DEATH_NOTIFICATION: {
18                         uint32_t target;
19                         void __user *cookie;
20                         struct binder_ref *ref;
21                         struct binder_ref_death *death;
22
23                         if (get_user(target, (uint32_t __user *)ptr))
24                                 return -EFAULT;
25                         ptr += sizeof(uint32_t);
26                         if (get_user(cookie, (void __user * __user *)ptr))
27                                 return -EFAULT;
28                         ptr += sizeof(void *);
29                         ref = binder_get_ref(proc, target);
30                         ......
31
32                         if (cmd == BC_REQUEST_DEATH_NOTIFICATION) {
33                                 ......
34                         } else {
35                                 if (ref->death == NULL) {
36                                         ......
37                                         break;
38                                 }
39                                 death = ref->death;
40                                 if (death->cookie != cookie) {
41                                        ......
42                                         break;
43                                 }
44                                 ref->death = NULL;
45                                 if (list_empty(&death->work.entry)) {
46                                         death->work.type = BINDER_WORK_CLEAR_DEATH_NOTIFICATION;
47                                         if (thread->looper & (BINDER_LOOPER_STATE_REGISTERED
| BINDER_LOOPER_STATE_ENTERED)) {
48                                                 list_add_tail(&death->work.entry, &thread->todo);
49                                         } else {
50                                                 list_add_tail(&death->work.entry, &proc->todo);
51                                                 wake_up_interruptible(&proc->wait);
52                                         }
53                                 } else {
54                                         BUG_ON(death->work.type != BINDER_WORK_DEAD_BINDER);
55                                         death->work.type = BINDER_WORK_DEAD_BINDER_AND_CLEAR;
56                                 }
57                         }
58                 } break;
59                 ......
60                 }
61                 *consumed = ptr - buffer;
62         }
63         return 0;
64 }
```

第23行和第26行分别将Client进程传进来的两个参数从用户空间缓冲区中复制出来，并且保存在

变量target和cookie中，它们分别对应于一个Binder代理对象的句柄值和地址值。第29行根据句柄值target找到对应的Binder引用对象，并保存在变量ref中。

前面在为一个Binder代理对象注册死亡接收通知时，Binder驱动程序将一个binder_ref_death结构体保存在了与该Binder代理对象对应的一个Binder引用对象的成员变量death中，因此，在为这个Binder代理对象注销死亡接收通知时，就应该将对应的Binder引用对象的成员变量death所指向的binder_ref_death结构体清理掉，如第44行代码所示。

第44行清理了用来描述死亡接收通知的binder_ref_death结构体之后，接下来第45行到第56行代码便向当前线程或者当前线程所属的Client进程的todo队列中添加一个类型为BINDER_WORK_CLEAR_DEATH_NOTIFICATION或者BINDER_WORK_DEAD_BINDER_AND_CLEAR的工作项，用来描述死亡接收通知注销操作的结果，并且返回给Client进程。

第45行的if语句判断binder_ref_death结构体death是否保存在某一个队列中。如果是，就表示它所对应的Binder本地对象已经死亡了。在前面的5.5.2小节中分析死亡接收通知的发送过程时提到，当一个Binder本地对象死亡时，Binder驱动程序会将一个类型为BINDER_WORK_DEAD_BINDER的工作项保存在目标Client进程的todo队列中。如果在Binder驱动程序将该工作项发送给目标Client进程处理之前，目标Client进程刚好又请求Binder驱动程序注销与该工作项对应的一个死亡接收通知，那么Binder驱动程序就会将死亡接收通知的发送操作和该死亡接收通知的注销结果返回操作合成一个类型为BINDER_WORK_DEAD_BINDER_AND_CLEAR的工作项交给目标Client进程来处理，这样就可以减少内核空间和用户空间的交互过程了，如第55行代码所示。

如果binder_ref_death结构体death没有保存在任何一个队列中，即第45行的if语句为true，那么Binder驱动程序就会将一个类型为BINDER_WORK_CLEAR_DEATH_NOTIFICATION的工作项添加到当前线程或者当前线程所属的Client进程的todo队列中，以便告诉Client进程它所请求的死亡接收通知注销操作已经完成了。如果当前线程是一个Binder线程，那么它的状态就应该为BINDER_LOOPER_STATE_REGISTERED 或者BINDER_LOOPER_STATE_ENTERED，即第47行的if语句为true，这时候Binder驱动程序就将该工作项添加到它的todo队列中来让它处理；否则，就添加到它所属的Client进程的todo队列中，并且唤醒相应的Binder线程来处理该工作项。

至此，一个死亡接收通知的注销过程就分析完成了。在接下来的5.6到5.9小节中，我们再结合四个使用情景更深入地分析Binder进程间通信机制的实现原理。

5.6 Service Manager的启动过程

Service Manager是Binder进程间通信机制的核心组件之一，它扮演着Binder进程间通信机制上下文管理者（Context Manager）的角色，同时负责管理系统中的Service组件，并且向Client组件提供获取Service代理对象的服务。

Service Manager运行在一个独立的进程中，因此，Service组件和Client组件也需要通过进程间通信机制来和它交互，而采用的进程间通信机制正好也是Binder进程间通信机制。从这个角度来看，Service Manager除了是Binder进程间通信机制的上下文管理者（Context Manager）之外，它也是一个特殊的Service组件。

Service Manager是由init进程负责启动的，而init进程是在系统启动时启动的，因此，Service Manager也是在系统启动时启动的，它的启动脚本如下所示。

system/core/rootdir/init.rc

```
1 service servicemanager /system/bin/servicemanager
2     user system
3     critical
4     onrestart restart zygote
5     onrestart restart media
```

第1行的关键字service表明Service Manager是以服务的形式启动的，它对应的程序文件和进程名称分别为/system/bin/servicemanager和“servicemanager”。第2行的关键字user表明Service Manager是以系统用户system的身份运行的。第3行的关键字critical表明Service Manager是系统中的一个关键服务。在系统的运行过程中，关键服务是不可以退出的，一旦退出，就会被系统重启，但是如果一个关键服务在4分钟内退出的次数大于4，那么系统就会重启，然后进入恢复模式。第4行和第5行的关键字onrestart表示一旦Service Manager被系统重启，也需要将zygote和media两个进程重新启动。

如前所述，Service Manager对应的程序文件为servicemanager，它的源代码目录结构如下：

~/Android/frameworks/base/cmds

```
----servicemanager
     ----binder.h
     ----binder.c
     ----service_manager.c
```

该程序的入口函数main实现在文件service_manager.c中，下面我们就从这个函数开始分析Service Manager的启动过程，它的实现如下所示。

frameworks/base/cmds/servicemanager/service_manager.c

```
01 void *svcmgr_handle;
02
03 int main(int argc, char **argv)
04 {
05     struct binder_state *bs;
06     void *svcmgr = BINDER_SERVICE_MANAGER;
07
08     bs = binder_open(128*1024);
09
10     if (binder_become_context_manager(bs)) {
11         LOGE("cannot become context manager (%s)\n", strerror(errno));
12         return -1;
13     }
14
15     svcmgr_handle = svcmgr;
16     binder_loop(bs, svcmgr_handler);
17     return 0;
18 }
```

Service Manager的启动过程由三个步骤组成：第一步是调用函数binder_open打开设备文件/dev/binder，以及将它映射到本进程的地址空间；第二步是调用函数binder_become_context_manager将自己注册为Binder进程间通信机制的上下文管理者；第三步是调用函数binder_loop来循环等待和处理Client进程的通信请求。

在继续分析Service Manager的启动过程之前，我们首先解释一下定义在Service Manager中的结构体binder_state和宏BINDER_SERVICE_MANAGER。

结构体binder_state的定义如下所示。

frameworks/base/cmds/servicemanager/binder.c

```
1 struct binder_state
2 {
3     int fd;
4     void *mapped;
5     unsigned mapsize;
6 };
```

Service Manager打开了设备文件/dev/binder之后，就会将得到的文件描述符保存在一个binder_state结构体的成员变量fd中，以便后面可以通过它来和Binder驱动程序交互。在前面的5.1.4小节中提到，一个进程如果要和Binder驱动程序交互，除了要打开设备文件/dev/binder之外，还需要将该设备文件映射到进程的地址空间，以便Binder驱动程序可以为它分配内核缓冲区来保存进程通信数据。因此，Service Manager就需要将设备文件/dev/binder映射到自己的进程地址空间，并且将映射后得到的地址空间大小和起始地址保存在一个binder_state结构体的成员变量mapsize和mapped中。

宏BINDER_SERVICE_MANAGER的定义如下所示。

frameworks/base/cmds/servicemanager/binder.h

```
1 /* the one magic object */
2 #define BINDER_SERVICE_MANAGER ((void*) 0)
```

前面提到，Service Manager是一个特殊的Service组件，它的特殊之处就在于与它对应的Binder本地对象是一个虚拟的对象。这个虚拟的Binder本地对象的地址值等于0，并且在Binder驱动程序中引用了它的Binder引用对象的句柄值也等于0。

回到函数main中，第6行将变量svcmgr的值设置为BINDER_SERVICE_MANAGER，接着第15行又将它保存在全局变量svcmgr_handle中，用来描述一个与Service Manager对应的Binder本地对象。

接下来，我们就分别分析函数binder_open、binder_become_context_manager和binder_loop的实现。

5.6.1 打开和映射Binder设备文件

函数binder_open用来打开设备文件/dev/binder，并且将它映射到进程的地址空间，它的实现如下所示。

frameworks/base/cmds/servicemanager/binder.c

```
01 struct binder_state *binder_open(unsigned mapsize)
02 {
03     struct binder_state *bs;
04
05     bs = malloc(sizeof(*bs));
06     if (!bs) {
07         errno = ENOMEM;
08         return 0;
09     }
10
11     bs->fd = open("/dev/binder", O_RDWR);
12     if (bs->fd < 0) {
13         fprintf(stderr,"binder: cannot open device (%s)\n",
14                 strerror(errno));
15         goto fail_open;
16     }
17
18     bs->mapsize = mapsize;
19     bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
20     if (bs->mapped == MAP_FAILED) {
21         fprintf(stderr,"binder: cannot map device (%s)\n",
22                 strerror(errno));
23         goto fail_map;
24     }
25
26         /* TODO: check version */
27
28     return bs;
29
30 fail_map:
```

```
31      close(bs->fd);
32  fail_open:
33      free(bs);
34      return 0;
35  }
```

从前面的调用过程可以知道，参数mapsize的大小为128 × 1024，即128K。

第11行调用函数open打开设备文件/dev/binder。在前面的5.1.3小节中提到，当使用进程函数open打开设备文件/dev/binder时，Binder驱动程序中的函数binder_open就会被调用，它会为当前进程创建一个binder_proc结构体，用来描述当前进程的Binder进程间通信状态。

第19行调用函数mmap将设备文件/dev/binder映射到进程的地址空间，它请求映射的地址空间大小为mapsize，即请求Binder驱动程序为进程分配128K大小的内核缓冲区。映射后得到的地址空间的起始地址和大小分别保存在一个binder_state结构体bs的成员变量mapped和mapsize中。最后，第28行将binder_state结构体bs返回给调用者，即函数main。

打开了设备文件/dev/binder，以及将它映射到进程的地址空间之后，Service Manager接下来就会将自己注册为Binder进程间通信机制的上下文管理者。

5.6.2　注册为Binder上下文管理者

Service Manager要成为Binder进程间通信机制的上下文管理者，就必须要通过IO控制命令BINDER_SET_CONTEXT_MGR将自己注册到Binder驱动程序中，这是通过调用函数binder_become_context_manager来实现的，它的定义如下所示。

frameworks/base/cmds/servicemanager/binder.c

```
1 int binder_become_context_manager(struct binder_state *bs)
2 {
3     return ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);
4 }
```

IO控制命令BINDER_SET_CONTEXT_MGR的定义如下所示。

kernel/goldfish/drivers/staging/android/binder.h

```
1 #define BINDER_SET_CONTEXT_MGR          _IOW('b', 7, int)
```

它只有一个整型参数，用来描述一个与Service Manager对应的Binder本地对象的地址值。由于与Service Manager对应的Binder本地对象是一个虚拟的对象，并且它的地址值等于0，因此，函数binder_become_context_manager就将IO控制命令BINDER_SET_CONTEXT_MGR的参数设置为0。

Binder驱动程序是在它的函数binder_ioctl中处理IO控制命令BINDER_SET_CONTEXT_MGR的，如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static struct binder_node *binder_context_mgr_node;
02 static uid_t binder_context_mgr_uid = -1;
03
04 static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
05 {
06         int ret;
07         struct binder_proc *proc = filp->private_data;
08         struct binder_thread *thread;
09         unsigned int size = _IOC_SIZE(cmd);
10         void __user *ubuf = (void __user *)arg;
11
12         ......
13
14         mutex_lock(&binder_lock);
```

```
15          thread = binder_get_thread(proc);
16          ......
17
18          switch (cmd) {
19          ......
20          case BINDER_SET_CONTEXT_MGR:
21                  if (binder_context_mgr_node != NULL) {
22                          printk(KERN_ERR "binder: BINDER_SET_CONTEXT_MGR already set\n");
23                          ret = -EBUSY;
24                          goto err;
25                  }
26                  if (binder_context_mgr_uid != -1) {
27                          if (binder_context_mgr_uid != current->cred->euid) {
28                                  printk(KERN_ERR "binder: BINDER_SET_"
29                                         "CONTEXT_MGR bad uid %d != %d\n",
30                                         current->cred->euid,
31                                         binder_context_mgr_uid);
32                                  ret = -EPERM;
33                                  goto err;
34                          }
35                  } else
36                          binder_context_mgr_uid = current->cred->euid;
37                  binder_context_mgr_node = binder_new_node(proc, NULL, NULL);
38                  if (binder_context_mgr_node == NULL) {
39                          ret = -ENOMEM;
40                          goto err;
41                  }
42                  binder_context_mgr_node->local_weak_refs++;
43                  binder_context_mgr_node->local_strong_refs++;
44                  binder_context_mgr_node->has_strong_ref = 1;
45                  binder_context_mgr_node->has_weak_ref = 1;
46                  break;
47          ......
48          }
49          ret = 0;
50  err:
51          if (thread)
52                  thread->looper &= ~BINDER_LOOPER_STATE_NEED_RETURN;
53          mutex_unlock(&binder_lock);
54          ......
55          return ret;
56  }
```

第7行首先获得前面Binder驱动程序为Service Manager进程创建的一个binder_proc结构体，并且保存在变量proc中，接着第15行调用函数binder_get_thread为当前线程创建一个binder_thread结构体。当前线程即为Service Manager进程的主线程，同时它也是Service Manager进程中的一个Binder线程。因此，在它第一次进入到Binder驱动程序时，Binder驱动程序就需要为它创建一个binder_thread结构体。

函数binder_get_thread的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static struct binder_thread *binder_get_thread(struct binder_proc *proc)
02 {
03          struct binder_thread *thread = NULL;
04          struct rb_node *parent = NULL;
05          struct rb_node **p = &proc->threads.rb_node;
06
07          while (*p) {
08                  parent = *p;
09                  thread = rb_entry(parent, struct binder_thread, rb_node);
10
11                  if (current->pid < thread->pid)
12                          p = &(*p)->rb_left;
```

```
13                  else if (current->pid > thread->pid)
14                          p = &(*p)->rb_right;
15                  else
16                          break;
17          }
18          if (*p == NULL) {
19                  thread = kzalloc(sizeof(*thread), GFP_KERNEL);
20                  if (thread == NULL)
21                          return NULL;
22                  ......
23                  thread->proc = proc;
24                  thread->pid = current->pid;
25                  init_waitqueue_head(&thread->wait);
26                  INIT_LIST_HEAD(&thread->todo);
27                  rb_link_node(&thread->rb_node, parent, p);
28                  rb_insert_color(&thread->rb_node, &proc->threads);
29                  thread->looper |= BINDER_LOOPER_STATE_NEED_RETURN;
30                  thread->return_error = BR_OK;
31                  thread->return_error2 = BR_OK;
32          }
33          return thread;
34  }
```

函数binder_get_thread在为一个线程创建一个binder_thread结构体之前，首先会检查与该线程所对应的binder_thread结构体是否已经存在。如果存在，就不用创建了，可直接将该binder_thread结构体返回给调用者。

在前面的5.1.1小节中介绍结构体binder_proc时提到，一个进程的所有Binder线程都保存在一个binder_proc结构体的成员变量threads所描述的一个红黑树中。由于这个红黑树是以线程的PID为关键字来组织的，因此，第7行到第17行的while循环就以当前线程的PID在这个红黑树中查找是否已经存在一个对应的binder_thread结构体。如果不存在，即最后得到的红黑树节点p为NULL，那么第19行到第31行代码就为当前线程创建一个binder_thread结构体，并且对它进行初始化，然后再将它添加到其宿主进程的成员变量threads所描述的一个红黑树中。

注意

Binder驱动程序为一个Binder线程创建一个binder_thread结构体时，会将它的状态设置为BINDER_LOOPER_STATE_NEED_RETURN，表示该线程在完成当前操作之后，需要马上返回到用户空间，而不可以去处理进程间的通信请求。

返回到函数binder_ioctl中，全局变量binder_context_mgr_node用来描述与Binder进程间通信机制的上下文管理者相对应的一个Binder实体对象，如果它不等于NULL，即第21行的if语句为true，那么就说明前面已经有组件将自己注册为Binder进程间通信机制的上下文管理者了。由于Binder驱动程序不允许重复注册Binder进程间通信机制的上下文管理者，因此，在这种情况下，它就直接出错返回了。

全局变量binder_context_mgr_uid用来描述注册了Binder进程间通信机制的上下文管理者的进程的有效用户ID，如果它的值不等于-1，即第26行的if语句为true，就说明前面已经有一个进程注册了Binder进程间通信机制的上下文管理者了。在这种情况下，Binder驱动程序就需要检查当前进程的有效用户ID是否等于全局变量binder_context_mgr_uid的值。如果不等于，即第27行的if语句为true，那么它就直接出错返回了。Binder驱动程序允许同一个进程重复使用IO控制命令BINDER_SET_CONTEXT_MGR来注册Binder进程间通信机制的上下文管理者，因为该进程前一次使用IO控制命令BINDER_SET_CONTEXT_MGR时，可能没有成功地将一个组件注册为Binder进程间通信机制的上下文管理者。例如，第37行调用函数binder_new_node创建一个Binder实体对象时，可能会失败。

通过了前面的两个合法性检查之后，第37行就可以调用函数binder_new_node为Service Manager创

建一个Binder实体对象了，并将它保存在全局变量binder_context_mgr_node中。函数binder_new_node的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static struct binder_node *
02 binder_new_node(struct binder_proc *proc, void __user *ptr, void __user *cookie)
03 {
04         struct rb_node **p = &proc->nodes.rb_node;
05         struct rb_node *parent = NULL;
06         struct binder_node *node;
07
08         while (*p) {
09                 parent = *p;
10                 node = rb_entry(parent, struct binder_node, rb_node);
11
12                 if (ptr < node->ptr)
13                         p = &(*p)->rb_left;
14                 else if (ptr > node->ptr)
15                         p = &(*p)->rb_right;
16                 else
17                         return NULL;
18         }
19
20         node = kzalloc(sizeof(*node), GFP_KERNEL);
21         if (node == NULL)
22                 return NULL;
23         ......
24         rb_link_node(&node->rb_node, parent, p);
25         rb_insert_color(&node->rb_node, &proc->nodes);
26         node->debug_id = ++binder_last_id;
27         node->proc = proc;
28         node->ptr = ptr;
29         node->cookie = cookie;
30         node->work.type = BINDER_WORK_NODE;
31         INIT_LIST_HEAD(&node->work.entry);
32         INIT_LIST_HEAD(&node->async_todo);
33         ......
34         return node;
35 }
```

第一个参数proc用来描述Service Manager进程；第二个参数ptr和第三个参数cookie用来描述一个Binder本地对象，它们分别指向该Binder本地对象内部的一个弱引用计数对象的地址值，以及该Binder本地对象的地址值。由于与Service Manager对应的Binder本地对象的地址值为0，因此，第二个参数ptr和第三个参数cookie的值均被指定为NULL。

在前面的5.1.1小节中介绍结构体binder_proc时提到，一个进程的所有Binder实体对象都保存在它的成员变量nodes所描述的一个红黑树中，并且以这些Binder实体对象的成员变量ptr为关键字。同时，在介绍结构体binder_node时又提到，一个Binder实体对象的成员变量ptr指向的是与它对应的Binder本地对象内部的一个弱引用计数对象的地址值，因此，第8行到第18行的while循环就以参数ptr为关键字，在这个红黑树中检查前面是否已经为参数ptr和cookie所描述的Binder本地对象创建过Binder实体对象。如果已经创建过，第17行就直接返回一个NULL值给调用者；否则，第20行到第32行代码就创建一个新的 Binder实体对象，并且对它进行初始化，最后将它加入到其宿主进程的成员变量nodes所描述的一个红黑树中。

返回到函数binder_ioctl中，第42行和第43行代码对刚才为Service Manager创建的Binder实体对象binder_context_mgr_node的内部弱引用计数local_weak_refs，以及内部强引用计数local_strong_refs进行初始化，即将它们的值设置为1，避免Binder驱动程序将其释放。接着第44行和第45行又将它的成员

变量has_strong_ref和has_weak_ref的值设置为1，表示Binder驱动程序已经请求Service Manager进程增加Service Manager组件的强引用计数和弱引用计数了。由于在系统运行期间，Service Manager组件是一定会存在的，因此，Binder驱动程序实际上不需要请求增加它的引用计数，而是直接将与它对应的Binder实体对象binder_context_mgr_node的成员变量has_strong_ref和has_weak_ref的值设置为1。

至此，Service Manager就成功地将自己注册为Binder进程间通信机制的上下文管理者了。函数binder_ioctl在返回到用户空间之前，第52行会将当前线程的状态位BINDER_LOOPER_STATE_NEED_RETURN清零，这样当该线程下次再进入到Binder驱动程序时，Binder驱动程序就可以将进程间通信请求分发给它处理了。

Service Manager返回到进程的用户空间之后，接着继续调用函数binder_loop来循环等待和处理Client进程的通信请求，即等待和处理Service组件的注册请求，以及其代理对象的获取请求。

5.6.3　循环等待Client进程请求

由于Service Manager需要在系统运行期间为Service组件和Client组件提供服务，因此，它就需要通过一个无限循环来等待和处理Service组件和Client组件的进程间通信请求，这是通过调用函数binder_loop来实现的，如下所示。

frameworks/base/cmds/servicemanager/binder.c

```
01 void binder_loop(struct binder_state *bs, binder_handler func)
02 {
03     int res;
04     struct binder_write_read bwr;
05     unsigned readbuf[32];
06 
07     bwr.write_size = 0;
08     bwr.write_consumed = 0;
09     bwr.write_buffer = 0;
10 
11     readbuf[0] = BC_ENTER_LOOPER;
12     binder_write(bs, readbuf, sizeof(unsigned));
13 
14     for (;;) {
15         bwr.read_size = sizeof(readbuf);
16         bwr.read_consumed = 0;
17         bwr.read_buffer = (unsigned) readbuf;
18 
19         res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
20 
21         if (res < 0) {
22             LOGE("binder_loop: ioctl failed (%s)\n", strerror(errno));
23             break;
24         }
25 
26         res = binder_parse(bs, 0, readbuf, bwr.read_consumed, func);
27         if (res == 0) {
28             LOGE("binder_loop: unexpected reply?!\n");
29             break;
30         }
31         if (res < 0) {
32             LOGE("binder_loop: io error %d %s\n", res, strerror(errno));
33             break;
34         }
35     }
36 }
```

从前面的调用过程可以知道，第一个参数bs指向前面在函数binder_open中创建的一个binder_state

结构体；第二个参数func指向Service Manager中的函数svcmgr_handler，它是用来处理Service组件和Client组件的进程间通信请求的。

在前面的5.1.1小节中介绍Binder命令协议BinderDriverCommandProtocol时提到，一个线程要通过协议BC_ENTER_LOOPER或者BC_REGISTER_LOOPER将自己注册为Binder线程，以便Binder驱动程序可以将进程间通信请求分发给它处理。由于Service Manager进程的主线程是主动成为一个Binder线程的，因此，它就使用BC_ENTER_LOOPER协议将自己注册到Binder驱动程序中。

第11行首先将BC_ENTER_LOOPER协议代码写入到缓冲区readbuf中，接着第12行调用函数binder_write将它发送到Binder驱动程序中。函数binder_write的实现如下所示。

frameworks/base/cmds/servicemanager/binder.c

```
01 int binder_write(struct binder_state *bs, void *data, unsigned len)
02 {
03     struct binder_write_read bwr;
04     int res;
05     bwr.write_size = len;
06     bwr.write_consumed = 0;
07     bwr.write_buffer = (unsigned) data;
08     bwr.read_size = 0;
09     bwr.read_consumed = 0;
10     bwr.read_buffer = 0;
11     res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
12     if (res < 0) {
13         fprintf(stderr,"binder_write: ioctl failed (%s)\n",
14                 strerror(errno));
15     }
16     return res;
17 }
```

由于BC_ENTER_LOOPER协议是通过IO控制命令BINDER_WRITE_READ发送到Binder驱动程序中的，而IO控制命令BINDER_WRITE_READ后面跟的参数是一个binder_write_read结构体，因此，第3行首先定义一个binder_write_read结构体bwr，接着第7行将参数data所指向的一块缓冲区作为它的输入缓冲区。第8行到第10行代码将binder_write_read结构体bwr的输出缓冲区设置为空，这样，当前线程将自己注册到Binder驱动程序中之后，就会马上返回到用户空间，而不会在Binder驱动程序中等待Client进程的通信请求。由于参数data所指向的一块缓冲区的内容已经被设置为BC_ENTER_LOOPER协议代码，因此，接下来第11行就可以调用函数ioctl将当前线程注册到Binder驱动程序中了。

IO控制命令BINDER_WRITE_READ是由Binder驱动程序中的函数binder_ioctl负责处理的，如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
02 {
03         int ret;
04         struct binder_proc *proc = filp->private_data;
05         struct binder_thread *thread;
06         unsigned int size = _IOC_SIZE(cmd);
07         void __user *ubuf = (void __user *)arg;
08
09         ......
10
11         mutex_lock(&binder_lock);
12         thread = binder_get_thread(proc);
13         ......
14
15         switch (cmd) {
16         case BINDER_WRITE_READ: {
```

```
17                  struct binder_write_read bwr;
18                  if (size != sizeof(struct binder_write_read)) {
19                          ret = -EINVAL;
20                          goto err;
21                  }
22                  if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {
23                          ret = -EFAULT;
24                          goto err;
25                  }
26                  ......
27                  if (bwr.write_size > 0) {
28                             ret = binder_thread_write(proc, thread, (void __user *)bwr.write_
buffer, bwr.write_size, &bwr.write_consumed);
29                          ......
30                  }
31                  if (bwr.read_size > 0) {
32                              ret = binder_thread_read(proc, thread, (void __user *)bwr.read_
buffer, bwr.read_size, &bwr.read_consumed, filp->f_flags & O_NONBLOCK);
33                          if (!list_empty(&proc->todo))
34                              wake_up_interruptible(&proc->wait);
35                          ......
36                  }
37                  ......
38                  if (copy_to_user(ubuf, &bwr, sizeof(bwr))) {
39                          ret = -EFAULT;
40                          goto err;
41                  }
42                  break;
43          }
44          ......
45          }
46          ret = 0;
47  err:
48          ......
49          return ret;
50  }
```

第12行调用函数binder_get_thread来获取与当前线程对应的一个binder_thread结构体，并保存在变量thread中。当前线程即为Service Manager进程的主线程，前面它将Service Manager注册为Binder进程间通信机制的上下文管理者时，Binder驱动程序已经为它创建过一个binder_thread结构体了，因此，第12行调用函数binder_get_thread时，就可以直接获得该binder_thread结构体。

第22行将从用户空间传进来的一个binder_write_read结构体复制出来，并且保存在变量bwr中。该结构体的输入缓冲区长度大于0，而输出缓冲区长度等于0，即第27行的if语为true，而第31行的if语句为false，因此，接下来第28行调用函数binder_thread_write处理完成BC_ENTER_LOOPER协议之后，就返回到用户空间了。

函数binder_thread_write处理BC_ENTER_LOOPER协议的相关代码如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 int
02 binder_thread_write(struct binder_proc *proc, struct binder_thread *thread,
03                      void __user *buffer, int size, signed long *consumed)
04 {
05          uint32_t cmd;
06          void __user *ptr = buffer + *consumed;
07          void __user *end = buffer + size;
```

```
08
09          while (ptr < end && thread->return_error == BR_OK) {
10                  if (get_user(cmd, (uint32_t __user *)ptr))
11                          return -EFAULT;
12                  ptr += sizeof(uint32_t);
13                  ......
14                  switch (cmd) {
15                  ......
16                  case BC_ENTER_LOOPER:
17                          ......
18                          if (thread->looper & BINDER_LOOPER_STATE_REGISTERED) {
19                                  thread->looper |= BINDER_LOOPER_STATE_INVALID;
20                                  binder_user_error("binder: %d:%d ERROR:"
21                                          " BC_ENTER_LOOPER called after "
22                                          "BC_REGISTER_LOOPER\n",
23                                          proc->pid, thread->pid);
24                          }
25                          thread->looper |= BINDER_LOOPER_STATE_ENTERED;
26                          break;
27                  ......
28                  }
29                  *consumed = ptr - buffer;
30          }
31          return 0;
32  }
```

它主要是将目标线程thread的状态设置为BINDER_LOOPER_STATE_ENTERED，以表明该线程是一个Binder线程，可以处理进程间通信请求。

注意

线程状态BINDER_LOOPER_STATE_REGISTERED和BINDER_LOOPER_STATE_ENTERED是互斥的，因此，第25行将线程状态设置为BINDER_LOOPER_STATE_ENTERED之前，第18行首先会检查该线程的状态是否已经为BINDER_LOOPER_STATE_REGISTERED。

函数binder_thread_write处理完成BC_ENTER_LOOPER协议之后，就返回到函数binder_ioctl中，后者接着又返回到用户空间中，即Service Manager进程的函数binder_write中，最后返回到函数binder_loop中。

在函数binder_loop中，第14行到第35行的循环不断使用IO控制命令BINDER_WRITE_READ来检查Binder驱动程序是否有新的进程间通信请求需要它来处理。如果有，就将它们交给函数binder_parse来处理；否则，当前线程就会在Binder驱动程序中睡眠等待，直到有新的进程间通信请求到来为止。

注意

在for循环中，每一次通过IO控制命令BINDER_WRITE_READ进入到Binder驱动程序时，所传递的binder_write_read结构体中的输入缓冲区的长度均为0，而输出缓冲区的长度等于缓冲区readbuf的大小，即128个字节。由于该binder_write_read结构体中的输入缓冲区的长度为0，因此，Binder驱动程序的函数binder_ioctl在处理这些IO控制命令BINDER_WRITE_READ时，只会调用函数binder_thread_read来检查Service Manager进程是否有新的进程间通信请求需要处理。

Binder驱动程序的函数binder_thread_read的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static int
02 binder_thread_read(struct binder_proc *proc, struct binder_thread *thread,
03         void  __user *buffer, int size, signed long *consumed, int non_block)
04 {
```

```
        void __user *ptr = buffer + *consumed;
        void __user *end = buffer + size;

        int ret = 0;
        int wait_for_proc_work;

        ......

retry:
        wait_for_proc_work = thread->transaction_stack == NULL && list_empty(&thread->todo);

        ......

        thread->looper |= BINDER_LOOPER_STATE_WAITING;
        if (wait_for_proc_work)
                proc->ready_threads++;
        mutex_unlock(&binder_lock);
        if (wait_for_proc_work) {
                ......
                binder_set_nice(proc->default_priority);
                if (non_block) {
                        if (!binder_has_proc_work(proc, thread))
                                ret = -EAGAIN;
                } else
                        ret = wait_event_interruptible_exclusive(proc->wait, binder_has_proc_
work(proc, thread));
        } else {
                if (non_block) {
                        if (!binder_has_thread_work(thread))
                                ret = -EAGAIN;
                } else
                        ret = wait_event_interruptible(thread->wait, binder_has_thread_work(thread));
        }
        mutex_lock(&binder_lock);
        if (wait_for_proc_work)
                proc->ready_threads--;
        thread->looper &= ~BINDER_LOOPER_STATE_WAITING;

        ......

        while (1) {
                ......
        }

done:

        *consumed = ptr - buffer;
        if (proc->requested_threads + proc->ready_threads == 0 &&
            proc->requested_threads_started < proc->max_threads &&
            (thread->looper & (BINDER_LOOPER_STATE_REGISTERED |
             BINDER_LOOPER_STATE_ENTERED)) /* the user-space code fails to */
             /*spawn a new thread if we leave this out */) {
                proc->requested_threads++;
                ......
                if (put_user(BR_SPAWN_LOOPER, (uint32_t __user *)buffer))
                        return -EFAULT;
        }
        return 0;
}
```

在前面的5.1.1小节中介绍结构体binder_thread和binder_transaction时提到，如果一个线程的事务堆栈transaction_stack不等于NULL，就表示它正在等待其他线程完成另外一个事务。另外，如果一个线程的todo队列不为空，就说明该线程有未处理的工作项。一个线程只有在其事务堆栈transaction_stack

为NULL，并且其todo队列为空时，才可以去处理其所属进程的todo队列中的待处理工作项；否则，它就要处理其事务堆栈transaction_stack中的事务或者todo队列中的待处理工作项。

第14行检查当前线程的事务堆栈transaction_stack是否为NULL，以及todo队列是否为空。如果两个条件都成立，那么就会将变量wait_for_proc_work的值设置为1，表示它接下来要检查它所属进程的todo队列中是否有未处理的工作项；否则，它接下来就要优先处理自己的事务或者工作项了。在继续往下执行之前，第18行首先将当前线程的状态设置为BINDER_LOOPER_STATE_WAITING，表示该线程正处于空闲状态，接着第19行判断变量wait_for_proc_work的值是否等于1。如果是，那么就说明当前线程所属的进程又多了一个空闲Binder线程，因此，第20行就将该进程的空闲Binder线程数ready_threads加1。接下来如果Binder驱动程序发现当前线程有新的工作项需要处理时，第40行就会将它的状态位BINDER_LOOPER_STATE_WAITING清空，并且根据变量wait_for_proc_work的值是否等于1，来决定是否需要减少它所属进程的空闲Binder线程数ready_threads，如第38行和第39行代码所示。

假设变量wait_for_proc_work的值等于1，即第22行的if语句为true，就说明当前线程需要检查它所属进程的todo队列中是否有未处理的工作项。在检查之前，第24行首先调用函数binder_set_nice把当前线程的优先级设置为它所属进程的优先级，这是因为如果它所属的进程有未处理的工作项，它就需要代表该进程去处理这个工作项。参数non_block表示当前线程是否以非阻塞模式打开设备文件/dev/binder。如果是，就表示当前线程不可以在Binder驱动程序中睡眠，即如果当前线程发现其所属进程的todo队列为空时，它不可以进入睡眠状态去等待该进程有新的未处理工作项，如第26行和第27行代码所示；否则，当前线程就会调用函数wait_event_interruptible_exclusive来睡眠等待直到其所属的进程有新的未处理工作项为止，如第29行代码所示。

判断一个进程是否有未处理的工作项是通过函数binder_has_proc_work来实现的，它的定义如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
1 static int
2 binder_has_proc_work(struct binder_proc *proc, struct binder_thread *thread)
3 {
4         return !list_empty(&proc->todo) || (thread->looper & BINDER_LOOPER_STATE_NEED_RETURN);
5 }
```

如果一个进程的todo队列不为空，那么就说明它有未处理的工作项，因此，函数的返回值就等于1。另外，如果当前线程的状态被设置为BINDER_LOOPER_STATE_NEED_RETURN，那么就表示当前线程需要马上返回到用户空间，因此，这时候函数的返回值也会等于1，防止当前线程进入睡眠等待状态。

返回到函数binder_thread_read中，假设变量wait_for_proc_work的值等于0，即第22行的if语句为false，那么就说明当前线程需要检查的是自己的todo队列中是否有未处理的工作项。同样，当前线程检查参数non_block的值是否等于1，如果是，那么当它发现自己的todo队列为空时，它就要马上返回到用户空间，如第32行和第33行代码所示；否则，它就要调用函数wait_event_interruptible_exclusive来睡眠等待直到它有新的未处理工作项为止，如第35行代码所示。

判断一个线程是否有未处理的工作项是通过函数binder_has_thread_work来实现的，它的定义如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
1 static int
2 binder_has_thread_work(struct binder_thread *thread)
3 {
4         return !list_empty(&thread->todo) || thread->return_error != BR_OK ||
5                 (thread->looper & BINDER_LOOPER_STATE_NEED_RETURN);
6 }
```

如果一个线程的todo队列不为空，那么就说明它有未处理的工作项，因此，函数的返回值就等于1。另外，如果该线程的状态被设置为BINDER_LOOPER_STATE_NEED_RETURN，那么就表示它需要马上返回到用户空间，因此，这时候函数的返回值也会等于1，防止它进入睡眠等待状态。还有另外一种情况也需要该线程马上返回到用户空间，即它上次在Binder驱动程序中处理某一个工作项时出现了错误，这时候Binder驱动程序就会将相应的错误代码（不等于BR_OK）设置到对应的binder_thread结构体的成员变量return_error中。因此，在这种情况下，函数的返回值也会等于1，防止它进入睡眠等待状态。

返回到函数binder_thread_read中，假设当前线程没有新的工作项需要处理，那么它要么睡眠在它所属进程的一个等待队列wait中，要么睡眠在自己的一个等待队列wait中，直到其他线程将它唤醒为止。

当前线程被唤醒之后，它就会调用第44行到第46行的while循环来处理它的工作项。处理完成之后，在返回到函数binder_ioctl之前，第51行到第60行代码还会检查是否需要请求当前线程所属的进程proc增加一个新的Binder线程来处理进程间通信请求。如果满足以下4个条件：

（1）进程proc的空闲线程数ready_threads等于0；

（2）Binder驱动程序当前不是正在请求进程proc增加一个新的Binder线程，即它的成员变量requested_threads的值等于0；

（3）Binder驱动程序请求进程proc增加的Binder线程数requested_threads_started小于预设的最大数max_threads。

（4）当前线程是一个已经注册了的Binder线程，即它的状态为BINDER_LOOPER_STATE_REGISTERED或者BINDER_LOOPER_STATE_ENTERED；

那么第58行就会将一个返回协议代码BR_SPAWN_LOOPER写入到用户空间缓冲区buffer中，以便进程proc可以创建一个新的线程加入到它的Binder线程池中。前面三个条件都比较容易理解，第4个条件是什么意思呢？如果当前线程不是一个Binder线程，那么它是不会去处理返回协议BR_SPAWN_LOOPER的，因此，这时候就不应该通过它来请求进程proc增加一个新的Binder线程。另一个需要注意的地方是，当Binder驱动程序请求进程proc增加一个新的Binder线程时，会将进程proc的成员变量requested_threads的值增加1，用来约束第2个条件，目的是避免同时请求进程proc增加多个Binder线程。当进程proc新增加的Binder线程注册到Binder驱动程序之后，Binder驱动程序就会将它的成员变量requested_threads的值减少1。

至此，Service Manager的启动过程就分析完成了。为了方便接下来内容的描述，我们假设Service Manager进程的主线程没有待处理的工作项，因此，它就睡眠在Binder驱动程序的函数binder_thread_read中，等待其他进程的Service组件或者Client组件向它发送进程间通信请求。

此外，从Service Manager的启动过程可以知道，当一个进程使用IO控制命令BINDER_WRITE_READ与Binder驱动程序交互时：

（1）如果传递给Binder驱动程序的binder_write_read结构体的输入缓冲区的长度大于0，那么Binder驱动程序就会调用函数binder_thread_write来处理输入缓冲区中的命令协议；

（2）如果传递给Binder驱动程序的binder_write_read结构体的输出缓冲区的长度大于0，那么Binder驱动程序就会调用函数binder_thread_read将进程需要处理的工作项写入到该输出缓冲区中，即将相应的返回协议写入到该输出缓冲区中，以便进程可以在用户空间处理。

在本章接下来的内容中，为了避免重复描述，当进程传递给Binder驱动程序的binder_write_read结构体的输入缓冲区的长度大于0时，我们就直接进入到函数binder_thread_write中分析输入缓冲区中的命令协议的处理过程。同样，当进程传递给Binder驱动程序的binder_write_read结构体的输出缓冲区的长度大

于0时，我们就直接进入到函数binder_thread_read中分析Binder驱动程序是如何将一个工作项，即一个进程间通信请求写入到输出缓冲区中的，以便进程返回到用户空间之后可以对它进行相应的处理。

5.7 Service Manager代理对象的获取过程

Service组件在启动时，需要将自己注册到Service Manager中；而Client组件在使用Service组件提供的服务之前，也需要通过Service Manager来获得Serivce组件的代理对象。由于Service Manager本身也是一个Service组件，因此，其他的Service组件和Client组件在使用它提供的服务之前，也需要先获得它的代理对象。作为一个特殊的Service组件，Service Manager代理对象的获取过程与其他的Service代理对象的获取过程有所不同，在本节中，我们将详细分析这个过程。

Service Manager代理对象的类型为BpServiceManager，它用来描述一个实现了IServiceManager接口的Client组件。根据前面5.2小节中的图5-16所描述的Client组件实现原理图，我们将图中的IXXXXXX接口和BpXXXXXX类直接替换为IServiceManager接口和BpServiceManager类，就可以得到Service Manager代理对象的类关系图，如图5-21所示。

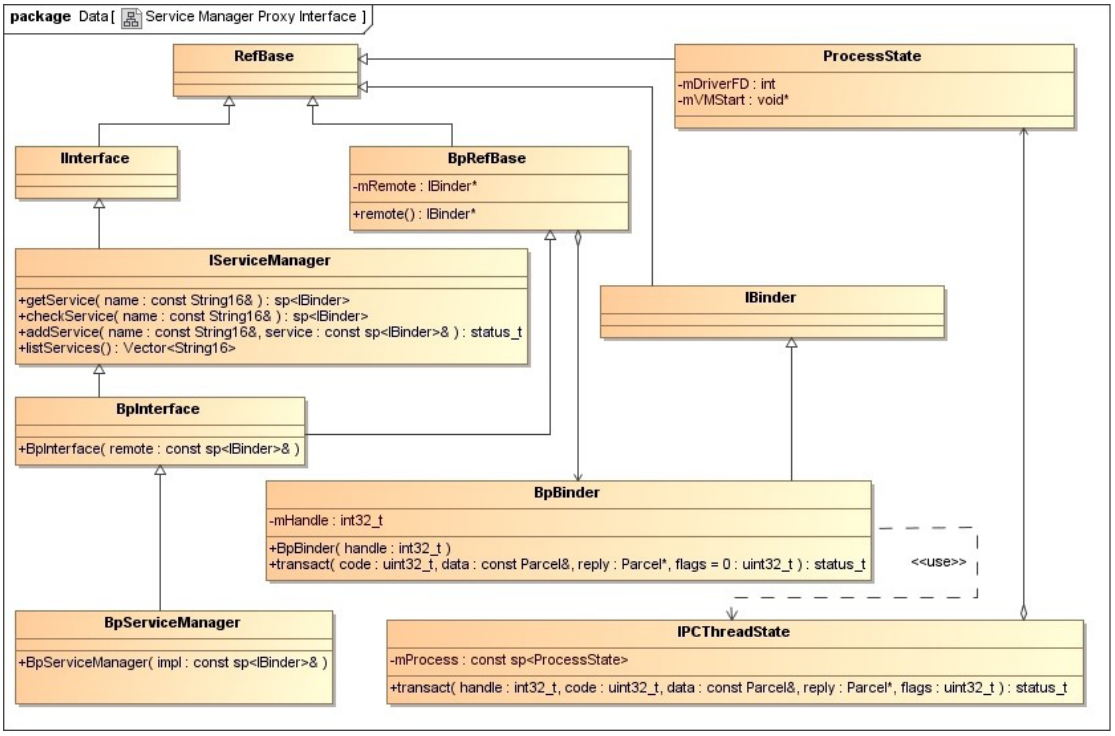


图5-21 Service Manager代理对象的类关系图

IServiceManager接口定义了四个成员函数getService、checkService、addService和listService，其中，getService和checkService用来获取Service组件的代理对象，addService用来注册Service组件，listService用来获取注册在Service Manager中的Service组件名称列表。

对于一般的Service组件来说，Client进程首先要通过Binder驱动程序来获得它的一个句柄值，然后才可以根据这个句柄创建一个Binder代理对象，最后将这个Binder代理对象封装成一个实现了特定接口的代理对象。由于Service Manager的句柄值恒为0，因此，获取它的一个代理对象的过程就省去了与Binder驱动程序交互的过程。

Android系统在应用程序框架层的Binder库中提供了一个函数defaultServiceManager来获得一个Service

Manager代理对象。例如，在前面5.3小节中介绍的应用程序FregClient调用函数defaultServiceManager来获得一个Service Manager代理对象的代码片段如下所示。

external/binder/client/FregClient.cpp

```
1 int main()
2 {
3         sp<IBinder> binder = defaultServiceManager()->getService(String16(FREG_SERVICE));
4         ......
5
6         return 0;
7 }
```

函数defaultServiceManager的实现如下所示。

frameworks/base/libs/binder/IServiceManager.cpp

```
01 sp<IServiceManager> defaultServiceManager()
02 {
03     if (gDefaultServiceManager != NULL) return gDefaultServiceManager;
04
05     {
06         AutoMutex _l(gDefaultServiceManagerLock);
07         if (gDefaultServiceManager == NULL) {
08             gDefaultServiceManager = interface_cast<IServiceManager>(
09                 ProcessState::self()->getContextObject(NULL));
10         }
11     }
12
13     return gDefaultServiceManager;
14 }
```

Binder库定义了gDefaultServiceManager和gDefaultServiceManagerLock两个全局变量，如下所示。

frameworks/base/libs/binder/Static.cpp

```
1 Mutex gDefaultServiceManagerLock;
2 sp<IServiceManager> gDefaultServiceManager;
```

全局变量gDefaultServiceManager是一个类型为IServiceManager的强指针，它指向进程内的一个BpServiceManager对象，即一个Service Manager代理对象；而全局变量gDefaultServiceManagerLock是一个互斥锁，是用来保证一个进程至多只有一个Service Manager代理对象的。结合锁机制来保证对象在进程中的唯一性，这是单例设计模式的经典实现。

在函数defaultServiceManager中，第3行首先检查全局变量gDefaultServiceManager是否不等于NULL。如果是，那么就说明Binder库已经为进程创建过一个Service Manager代理对象了，因此，第3行代码就可以直接将它返回给调用者；否则，第8行和第9行代码就创建一个Service Manager代理对象，并且将它保存在全局变量gDefaultServiceManager中，最后第13行将它返回给调用者。

第8行和第9行代码包含了三次函数调用。第一次是调用ProcessState类的静态成员函数self获得进程内的一个ProcessState对象，第二次是调用前面获得的ProcessState对象的成员函数getContextObject创建一个Binder代理对象，第三次是调用模板函数interface_cast<IServiceManager>将前面获得的Binder代理对象封装成一个Service Manager代理对象。下面我们就分别分析这三个函数的实现。

ProcessState类的静态成员函数self的实现如下所示。

frameworks/base/libs/binder/ProcessState.cpp

```
1 sp<ProcessState> ProcessState::self()
2 {
3     if (gProcess != NULL) return gProcess;
4
5     AutoMutex _l(gProcessMutex);
6     if (gProcess == NULL) gProcess = new ProcessState;
7     return gProcess;
8 }
```

在Binder库中，定义了gProcess和gProcessMutex两个全局变量，如下所示。

frameworks/base/libs/binder/Static.cpp

```
1 Mutex gProcessMutex;
2 sp<ProcessState> gProcess;
```

全局变量gProcess是一个类型为ProcessState的强指针，它指向进程内的一个ProcessState对象；而全局变量gProcessMutex是一个互斥锁，是用来保证一个进程至多只有一个ProcessState对象的，这同样是一个单例设计模式。

在ProcessState类的静态成员函数self中，第3行首先检查全局变量gProcess是否不等于NULL。如果是，那么就说明Binder库已经为进程创建过一个ProcessState了，因此，第3行代码就可以直接将它返回给调用者；否则，第6行代码就直接创建一个ProcessState对象，并且将它保存在全局变量gProcess中，最后第7行将它返回给调用者。

进程中的ProcessState对象的创建过程如下所示。

frameworks/base/libs/binder/ProcessState.cpp

```
01 ProcessState::ProcessState()
02     : mDriverFD(open_driver())
03     , mVMStart(MAP_FAILED)
04     ......
05 {
06     if (mDriverFD >= 0) {
07         ......
08         // mmap the binder, providing a chunk of virtual address space to receive transactions.
09         mVMStart = mmap(0, BINDER_VM_SIZE, PROT_READ, MAP_PRIVATE | MAP_NORESERVE, mDriverFD, 0);
10         ......
11     }
12     ......
13 }
```

在ProcessState类的构造函数中，第2行首先调用函数open_driver打开设备文件/dev/binder，并且将得到的文件描述符保存在成员变量mDriverFD中。接着第9行调用函数mmap把设备文件/dev/binder映射到进程的地址空间，映射的地址空间大小为BINDER_VM_SIZE，它的定义如下所示。

frameworks/base/libs/binder/ProcessState.cpp

```
1 #define BINDER_VM_SIZE ((1*1024*1024) - (4096 *2))
```

从前面5.1.4小节的内容可以知道，将设备文件/dev/binder映射到进程的地址空间实际上是请求Binder驱动程序为进程分配内核缓冲区，这个内核缓冲区的大小被Binder库默认设置为1016KB。

接下来，我们重点分析函数open_driver的实现，它的定义如下所示。

frameworks/base/libs/binder/ProcessState.cpp

```
01 static int open_driver()
02 {
03     ......
04
05     int fd = open("/dev/binder", O_RDWR);
06     if (fd >= 0) {
07         ......
08         status_t result = ioctl(fd, BINDER_VERSION, &vers);
09         ......
10
11         if (result != 0 || vers != BINDER_CURRENT_PROTOCOL_VERSION) {
12             LOGE("Binder driver protocol does not match user space protocol!");
13             close(fd);
14             fd = -1;
15         }
16
```

```
17          ......
18          size_t maxThreads = 15;
19          result = ioctl(fd, BINDER_SET_MAX_THREADS, &maxThreads);
20          ......
21
22      } else {
23          LOGW("Opening '/dev/binder' failed: %s\n", strerror(errno));
24      }
25      return fd;
26 }
```

第5行首先调用函数open打开设备文件/dev/binder，接着第8行使用IO控制命令BINDER_VERSION来获得Binder驱动程序的版本号，最后第19行使用IO控制命令BINDER_SET_MAX_THREADS来通知Binder驱动程序，它最多可以请求进程创建15个Binder线程来处理进程间通信请求。

获得了进程的ProcessState对象之后，回到函数defaultServiceManager中，接着就调用它的成员函数getContextObject来创建一个Binder代理对象。

ProcessState类的成员函数getContextObject的实现如下所示。

frameworks/base/libs/binder/ProcessState.cpp

```
1 sp<IBinder> ProcessState::getContextObject(const sp<IBinder>& caller)
2 {
3     if (supportsProcesses()) {
4         return getStrongProxyForHandle(0);
5     } else {
6         return getContextObject(String16("default"), caller);
7     }
8 }
```

第3行调用成员函数supportsProcesses来检查系统是否支持Binder进程间通信机制，即检查进程是否成功地打开了设备文件/dev/binder，它的实现如下所示。

frameworks/base/libs/binder/ProcessState.cpp

```
1 bool ProcessState::supportsProcesses() const
2 {
3     return mDriverFD >= 0;
4 }
```

回到ProcessState 类的成员函数getContextObject中，假设系统是支持Binder进程间通信机制的，因此，它接着就会调用成员函数getStrongProxyForHandle来创建一个Binder代理对象，它的实现如下所示。

frameworks/base/libs/binder/ProcessState.cpp

```
01 sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
02 {
03     sp<IBinder> result;
04
05     AutoMutex _l(mLock);
06
07     handle_entry* e = lookupHandleLocked(handle);
08
09     if (e != NULL) {
10         // We need to create a new BpBinder if there isn't currently one, OR we
11         // are unable to acquire a weak reference on this current one.  See comment
12         // in getWeakProxyForHandle() for more info about this.
13         IBinder* b = e->binder;
14         if (b == NULL || !e->refs->attemptIncWeak(this)) {
15             b = new BpBinder(handle);
16             e->binder = b;
17             if (b) e->refs = b->getWeakRefs();
18             result = b;
19         } else {
20             // This little bit of nastyness is to allow us to add a primary
```

```
21              // reference to the remote proxy when this team doesn't have one
22              // but another team is sending the handle to us.
23              result.force_set(b);
24              e->refs->decWeak(this);
25          }
26      }
27
28      return result;
29  }
```

从前面的调用过程可以知道，参数handle的值等于0，表示要创建的Binder代理对象的句柄值等于0，即要创建一个Service Manager代理对象。

在前面的5.4.4小节中介绍Binder代理对象的生命周期时提到，Binder库为每一个进程维护了一个handle_entry类型的Binder代理对象列表，它以句柄值作为关键字来维护进程内部所有的Binder代理对象。这个Binder代理对象列表保存在ProcessState类的成员变量mHandleToObject中，它的定义如下所示。

frameworks/base/include/binder/ProcessState.h

```
01 class ProcessState : public virtual RefBase
02 {
03              ......
04
05 private:
06              ......
07              struct handle_entry {
08                  IBinder* binder;
09                  RefBase::weakref_type* refs;
10              };
11
12              ......
13              Vector<handle_entry>mHandleToObject;
14
15              ......
16 };
```

每一个Binder代理对象都使用一个handle_entry结构体来描述。结构体handle_entry的两个成员变量binder和refs分别指向一个Binder代理对象，以及它内部的一个弱引用计数对象。

回到ProcessState类的成员函数getContextObject中，第7行首先调用成员函数lookupHandleLocked来检查成员变量mHandleToObject中是否已经存在一个与句柄值handle对应的handle_entry结构体，它的实现如下所示。

frameworks/base/libs/binder/ProcessState.cpp

```
01 ProcessState::handle_entry* ProcessState::lookupHandleLocked(int32_t handle)
02 {
03      const size_t N=mHandleToObject.size();
04      if (N <= (size_t)handle) {
05          handle_entry e;
06          e.binder = NULL;
07          e.refs = NULL;
08          status_t err = mHandleToObject.insertAt(e, N, handle+1-N);
09          if (err < NO_ERROR) return NULL;
10      }
11      return &mHandleToObject.editItemAt(handle);
12 }
```

一个Binder代理对象的句柄值同时也是它在列表mHandleToObject中的索引值。第4行首先检查句柄值handle的值是否大于或者等于列表mHandleToObject的大小。如果是，即第4行的if语句为true，那么mHandleToObject列表里面就不存在一个与句柄值handle对应的handle_entry结构体，接下来第5行到第8行代码就会在列表mHandleToObject的第*N*到第handle个位置上分别插入一个handle_entry结构体，最后第11行就可以将与句柄值handle对应的handle_entry结构体返回给调用者。

注意

这些插入的handle_entry结构体与Binder代理对象都还没有关联起来，因此，它们的成员变量binder和refs均被初始化为NULL。如果句柄值handle的值小于列表mHandleToObject的大小，那么第11行就可以直接将与它对应的一个handle_entry结构体返回给调用者了。

回到ProcessState 类的成员函数getStrongProxyForHandle中，第7行将前面从Binder代理对象列表中获得的handle_entry结构体保存在变量e中。

如果handle_entry结构体e的成员变量binder的值等于NULL，即第14行的if语句为true，就说明进程尚未为句柄值handle创建过Binder代理对象，因此，第15行就会根据句柄值handle创建一个Binder代理对象，并且将它保存在handle_entry结构体e的成员变量binder中。如果handle_entry结构体e的成员变量binder的值不为NULL，即它已经指向了一个Binder代理对象，就需要继续检查这个Binder代理对象是否还活着，方法是调用它的弱引用计数对象的成员函数attemptIncWeak来尝试增加它的弱引用计数。由于Binder代理对象（即BpBinder对象）的生命周期是受弱引用计数控制的，因此如果不能成功增加它的弱引用计数，那么就说明它已经被销毁了。在这种情况下，也需要重新为句柄值handle创建一个Binder代理对象。

如果handle_entry结构体e的成员变量binder的值不等于NULL，并且调用它的成员变量refs的成员函数attemptIncWeak能够成功地增加与它对应的一个Binder代理对象的弱引用计数，即第14行的if语句为false，那么就说明在进程中已经有一个Binder代理对象与句柄值handle相对应，并且这个Binder代理对象是有效的，因此，可以直接将它返回给调用者。

注意

函数在将这个Binder代理对象返回给调用者之前，第24行会减少它的弱引用计数，因为前面第14行增加了它的弱引用计数。

获得了与句柄值0对应的Binder代理对象之后，回到函数defaultServiceManager中，接着就调用模板函数interface_cast<IServiceManager>将这个Binder代理对象封装为一个Service Manager代理对象。

模板函数interface_cast<IServiceManager>的实现如下所示。

frameworks/base/include/binder/IInterface.h

```
1 template<typename INTERFACE>
2 inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
3 {
4     return INTERFACE::asInterface(obj);
5 }
```

模板参数INTERFACE指向IServiceManager接口，因此，它实际上是调用IServiceManager接口的成员函数asInterface，将一个句柄值为0的Binder代理对象封装为一个Service Manager代理对象。

IServiceManager接口的成员函数asInterface是通过宏IMPLEMENT_META_INTERFACE来定义的，如下所示。

frameworks/base/libs/binder/IServiceManager.cpp

```
1 IMPLEMENT_META_INTERFACE(ServiceManager, "android.os.IServiceManager");
```

将宏IMPLEMENT_META_INTERFACE展开，就可以得到IServiceManager接口的成员函数asInterface，如下所示。

frameworks/base/libs/binder/IServiceManager.cpp

```
01 android::sp<IServiceManager> IServiceManager::asInterface(
02       const android::sp<android::IBinder>& obj)
03 {
04         android::sp<IServiceManager> intr;
```

```
05          if (obj != NULL) {
06              intr = static_cast<IServiceManager*>(
07                  obj->queryLocalInterface(
08                          IServiceManager::descriptor).get());
09              if (intr == NULL) {
10                  intr = new BpServiceManager(obj);
11              }
12          }
13          return intr;
14  }
```

从前面的调用过程可以知道，参数obj指向一个Binder代理对象，即一个BpBinder对象，它的成员函数queryLocalInterface的返回值为NULL。因此，最后会执行第10行代码创建一个Service Manager代理对象，即一个BpServiceManager对象，并且将它的IServiceManager接口返回给调用者。

至此，一个Service Manager代理对象的获取过程就分析完成了。有了这个Service Manager代理对象之后，Service组件就可以在启动的过程中使用它的成员函数addService将自己注册到Service Manager中，而Client组件就可以使用它的成员函数getService来获得一个指定名称的Service组件的代理对象了。在接下来的5.8和5.9小节中，我们继续分析Service组件的启动过程及其代理对象的获取过程。

5.8 Service组件的启动过程

Service组件是在Server进程中运行的。Server进程在启动时，会首先将它里面的Service组件注册到Service Manager中，接着再启动一个Binder线程池来等待和处理Client进程的通信请求。在本节中，我们将以5.3小节介绍的Service组件FregService为例，详细分析一个Service组件的启动过程。

前面5.2小节的图5-15描述了Service组件的实现原理，我们将图中的IXXXXXX接口及BnXXXXXX和XXXXXX类直接替换为IFregService接口及BnFregService和FregService类，就可以得到Service组件FregService的类关系图，如图5-22所示。

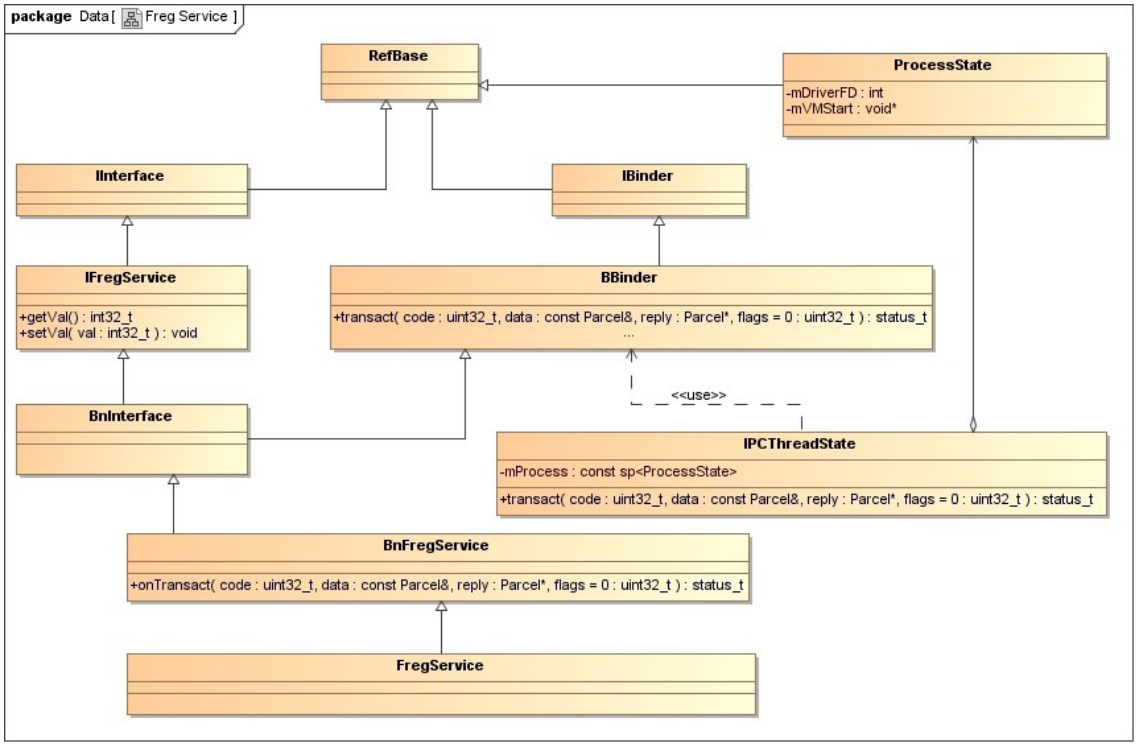


图5-22 Service组件FregService的类关系图

Service组件FregService是运行在FregServer进程中的，FregServer进程的启动过程如下所示。

external/binder/server/FregServer.cpp

```
1 int main(int argc, char** argv)
2 {
3         FregService::instantiate();
4
5         ProcessState::self()->startThreadPool();
6         IPCThreadState::self()->joinThreadPool();
7
8         return 0;
9 }
```

第3行调用FregService类的静态成员函数instantiate创建一个FregService组件，并且将它注册到Service Manager中；接着第5行启动了一个Binder线程池；最后第6行将主线程也加入到这个Binder线程池中去等待和处理Client进程的通信请求。

在接下来的内容中，我们先分析Service组件FregService的注册过程，然后再分析FregServer进程是如何启动一个Binder线程池来等待和处理Client进程的通信请求的。

5.8.1 注册Service组件

在将Service组件注册到Service Manager之前，我们首先要获得一个Service Manager代理对象，然后才可以通过它将该Service组件注册到Service Manager中。

以Service组件FregService的注册过程为例，FregService类的静态成员函数instantiate首先调用Binder库提供的函数defaultServiceManager来获得一个Service Manager代理对象，然后再调用它的成员函数addService将该Service组件FregService注册到Service Manager中，它的实现如下所示。

external/binder/server/FregServer.cpp

```
01 class FregService : public BnFregService
02 {
03 ......
04
05 public:
06         static void instantiate()
07         {
08                 defaultServiceManager()->addService(String16(FREG_SERVICE), new FregService());
09         }
10
11 ......
12 };
```

函数defaultServiceManager返回的Service Manager代理对象的类型为BpServiceManager，因此，FregService类的静态成员函数instantiate实际上是调用了BpServiceManager类的成员函数addService来注册Service组件FregService，它的实现如下所示。

frameworks/base/libs/binder/IServiceManager.cpp

```
01 class BpServiceManager : public BpInterface<IServiceManager>
02 {
03 public:
04     ......
05
06     virtual status_t addService(const String16& name, const sp<IBinder>& service)
07     {
08         Parcel data, reply;
09         data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
10         data.writeString16(name);
11         data.writeStrongBinder(service);
```

```
12          status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data, &reply);
13          return err == NO_ERROR ? reply.readExceptionCode() : err;
14      }
15
16      ......
17  };
```

Client进程和Server进程的一次进程间通信过程可以划分为如下五个步骤。

（1）Client进程将进程间通信数据封装成一个Parcel对象，以便可以将进程间通信数据传递给Binder驱动程序。

（2）Client进程向Binder驱动程序发送一个BC_TRANSACTION命令协议。Binder驱动程序根据协议内容找到目标Server进程之后，就会向Client进程发送一个BR_TRANSACTION_COMPLETE返回协议，表示它的进程间通信请求已经被接受。Client进程接收到Binder驱动程序发送给它的BR_TRANSACTION_COMPLETE返回协议，并且对它进行处理之后，就会再次进入到Binder驱动程序中去等待目标Server进程返回进程间通信结果。

（3）Binder驱动程序在向Client进程发送BR_TRANSACTION_COMPLETE返回协议的同时，也会向目标Server进程发送一个BR_TRANSACTION返回协议，请求目标Server进程处理该进程间通信请求。

（4）Server进程接收到Binder驱动程序发来的BR_TRANSACTION返回协议，并且对它进行处理之后，就会向Binder驱动程序发送一个BC_REPLY命令协议。Binder驱动程序根据协议内容找到目标Client进程之后，就会向Server进程发送一个BR_TRANSACTION_COMPLETE返回协议，表示它返回的进程间通信结果已经收到了。Server进程接收到Binder驱动程序发送给它的BR_TRANSACTION_COMPLETE返回协议，并且对它进行处理之后，一次进程间通信过程就结束了。接着它会再次进入到Binder驱动程序中去等待下一次进程间通信请求。

（5）Binder驱动程序向Server进程发送BR_TRANSACTION_COMPLETE返回协议的同时，也会向目标Client进程发送一个BR_REPLY返回协议，表示Server进程已经处理完成它的进程间通信请求了，并且将进程间通信结果返回给它。

注意

在第2步和第4步中，为什么Binder驱动程序要发送一个BR_TRANSACTION_COMPLETE返回协议来响应Client进程和Server进程的BC_TRANSACTION与BC_REPLY命令协议呢？Client进程和Server进程其实并不需要对BR_TRANSACTION_COMPLETE返回协议做特殊的处理。这样做其实是为了让Client进程和Server进程在执行进程间通信的过程中，有机会返回到用户空间去做其他事情，从而增加它们的并发处理能力。

在本节接下来的内容中，我们将根据上述五个步骤来分析BpServiceManager类的成员函数addService的实现。

5.8.1.1 封装进程间通信数据

在BpServiceManager类的成员函数addService中，第9行到第11行代码分别将进程间通信数据写入到一个Parcel对象data中，然后将Parcel对象data的内容传递给Binder驱动程序。

第9行首先调用Parcel对象data的成员函数writeInterfaceToken写入一个Binder进程间通信请求头，它的实现如下所示。

frameworks/base/libs/binder/Parcel.cpp

```
1 status_t Parcel::writeInterfaceToken(const String16& interface)
2 {
3     writeInt32(IPCThreadState::self()->getStrictModePolicy() |
```

```
4                 STRICT_MODE_PENALTY_GATHER);
5     // currently the interface identification token is just its name as a string
6     return writeString16(interface);
7 }
```

Binder进程间通信请求头由两部分内容组成。第一部分内容是一个整数值，用来描述一个Strict Mode Policy；第二部分内容是一个字符串，用来描述所请求服务的接口描述符。

在Android应用程序中，我们可以为线程定义一些Strict Mode Policy。如果线程在运行过程中违反了这些预先定义好的Strict Mode Policy，那么系统就会发出警告。这样开发人员就方便调试应用程序的功能，并且可以对它们进行优化。例如，我们可以禁止在应用程序的UI线程中执行写磁盘操作，因为写磁盘是一个耗时的操作，会影响用户体验。这时候我们就可以为应用程序的UI线程定义一个不允许写磁盘的Strict Mode Policy。当开发人员不小心在应用程序的UI线程中执行了一个写磁盘操作时，系统就会发出警告，这样开发人员就可以根据这个警告来对应用程序进行优化。

当Client进程中的一个线程将一个进程间通信请求发送给Server进程的某一个线程处理时，其实是要求这个Server线程代替它来执行某一个操作，因此，Binder进程间通信机制就会尽量地将Server线程模拟成Client线程来执行这个操作，这样就要求将Client线程的某些属性设置到Server线程中，Strict Mode Policy便是其中的一个属性。因此，第3行和第4行代码首先获得当前线程的Strict Mode Policy，然后将它与宏STRICT_MODE_PENALTY_GATHER执行一个按位或操作，最后将得到的结果写入到Parcel对象data中去传递给目标Server线程。STRICT_MODE_PENALTY_GATHER是用来描述一个线程的Strict Mode Policy的，它用于Binder进程间通信机制，表示Server线程在处理进程间通信请求时，即使违反了预先设定的Strict Mode Policy，系统也不应该发出警告，而是把这些警告收集起来，最后统一发送给Client线程来处理。

第6行将一个服务接口描述符写入到Parcel对象data中。Server组件在处理一个进程间通信请求时，会将这个服务接口的描述符读取出来，并且验证它是否是一个合法的服务接口描述符，即是否是自己所实现的服务接口的描述符。如果不是，则说明这是一个非法的进程间通信请求。

回到BpServiceManager类的成员函数addService中，接着第10行调用Parcel对象data的成员函数writeString16写入将要注册的Service组件的名称。

第11行调用Parcel对象data的成员函数writeStrongBinder将要注册的Service组件封装成一个flat_binder_object结构体，然后传递给Binder驱动程序。将一个Service组件封装成一个flat_binder_object结构体是Binder进程间通信过程的一个重要步骤，接下来我们就着重分析这个过程。

Parcel类的成员函数writeStrongBinder的实现如下所示。

frameworks/base/libs/binder/Parcel.cpp

```
1 status_t Parcel::writeStrongBinder(const sp<IBinder>& val)
2 {
3     return flatten_binder(ProcessState::self(), val, this);
4 }
```

第3行调用全局函数flatten_binder将一个Service组件封装成一个flat_binder_object结构体，它的实现如下所示。

frameworks/base/libs/binder/Parcel.cpp

```
01 status_t flatten_binder(const sp<ProcessState>& proc,
02     const sp<IBinder>& binder, Parcel* out)
03 {
04     flat_binder_object obj;
05
06     obj.flags = 0x7f | FLAT_BINDER_FLAG_ACCEPTS_FDS;
07     if (binder != NULL) {
08         IBinder *local = binder->localBinder();
```

```
09          if (!local) {
10              ......
11          } else {
12              obj.type = BINDER_TYPE_BINDER;
13              obj.binder = local->getWeakRefs();
14              obj.cookie = local;
15          }
16      } else {
17          ......
18      }
19
20      return finish_flatten_binder(binder, obj, out);
21  }
```

第4行首先定义一个flat_binder_object结构体obj，接着第6行将它的标志值设置为（0x7f | FLAT_BINDER_FLAG_ACCEPTS_FDS）。其中，0x7f用来描述将要注册的Service组件在处理一个进程间通信请求时，它所使用的Server线程的优先级不能低于0x7f；而FLAT_BINDER_FLAG_ACCEPTS_FDS表示可以将包含文件描述符的进程间通信数据传递给将要注册的Service组件处理。

从前面的调用过程可以知道，参数binder指向的是一个FregService组件，它继承了BBinder类。BBinder类是用来描述一个Binder本地对象的，它的成员函数localBinder用来返回一个Binder本地对象接口。由于FregService组件是一个Binder本地对象，因此，第8行得到的变量local的值不等于NULL。接下来第12行到第14行代码就将flat_binder_object结构体obj设置为一个BINDER_TYPE_BINDER类型的Binder对象，并且将它的成员变量cookie和ptr的值分别设置为Binder本地对象local的地址值及其内部的一个弱引用计数对象的地址值。

最后，第20行调用全局函数finish_flatten_binder将flat_binder_object结构体obj写入到Parcel对象out中，它的实现如下所示。

frameworks/base/libs/binder/Parcel.cpp

```
1 inline static status_t finish_flatten_binder(
2     const sp<IBinder>& binder, const flat_binder_object& flat, Parcel* out)
3 {
4     return out->writeObject(flat, false);
5 }
```

第4行调用Parcel对象out的成员函数writeObject将flat_binder_object结构体flat写入到内部。

Parcel类的成员函数writeObject的实现如下所示。

frameworks/base/libs/binder/Parcel.cpp

```
01 status_t Parcel::writeObject(const flat_binder_object& val, bool nullMetaData)
02 {
03     const bool enoughData = (mDataPos+sizeof(val)) <= mDataCapacity;
04     const bool enoughObjects = mObjectsSize < mObjectsCapacity;
05     if (enoughData && enoughObjects) {
06 restart_write:
07         *reinterpret_cast<flat_binder_object*>(mData+mDataPos) = val;
08
09         // Need to write meta-data?
10         if (nullMetaData || val.binder != NULL) {
11             mObjects[mObjectsSize] = mDataPos;
12             acquire_object(ProcessState::self(), val, this);
13             mObjectsSize++;
14         }
15
16         // remember if it's a file descriptor
17         if (val.type == BINDER_TYPE_FD) {
18             mHasFds = mFdsKnown = true;
19         }
20
```

```
21          return finishWrite(sizeof(flat_binder_object));
22      }
23
24      if (!enoughData) {
25          const status_t err = growData(sizeof(val));
26          if (err != NO_ERROR) return err;
27      }
28      if (!enoughObjects) {
29          size_t newSize = ((mObjectsSize+2)*3)/2;
30          size_t* objects = (size_t*)realloc(mObjects, newSize*sizeof(size_t));
31          if (objects == NULL) return NO_MEMORY;
32          mObjects = objects;
33          mObjectsCapacity = newSize;
34      }
35
36      goto restart_write;
37  }
```

Parcel类内部有mData和mObjects两个缓冲区，其中，mData是一个数据缓冲区，它里面的内容可能包含有整数、字符串或者Binder对象，即flat_binder_object结构体；mObjects是一个偏移数组，它保存了在数据缓冲区mData中的所有Binder对象的位置。Binder驱动程序就是通过这个偏移数组找到进程间通信数据中的Binder对象的，以便对它们进行特殊处理。

Parcel类的成员变量mDataPos记录了数据缓冲区mData下一个用来写入数据的位置，而成员变量mDataCapacity则记录了数据缓冲区mData的总大小，因此，第3行代码用来判断数据缓冲区mData是否还有足够的空间来写入一个Binder对象。

Parcel类的成员变量mObjectsSize记录了偏移数组mObjects下一个用来写入数据的位置，而成员变量mObjectsCapacity则记录了偏移数组mObjects的总大小，因此，第4行代码用来判断偏移数组mObjects是否还有足够的空间来写入一个Binder对象的偏移位置。

如果数据缓冲区mData和偏移数组mObjects还有足够的空间来写入一个Binder对象，即第5行的if语句为true，那么第7行到第14行代码就将参数val所描述的Binder对象写入到数据缓冲区mData和偏移数组mObjects中。

如果数据缓冲区mData或者偏移数组mObjects的空间不足以用来写入一个Binder对象，即第5行的if语句为false，那么第24行到第34行代码就会首先扩展它们的空间，然后第36行再跳转到第7行到第14行代码将参数val所描述的Binder对象写入到数据缓冲区mData和偏移数组mObjects中。

第17行判断刚才写入的Binder对象的类型是否为BINDER_TYPE_FD。如果是，那么就说明该Parcel对象的数据缓冲区中包含有文件描述符，因此，第18行将成员变量mHasFds 和mFdsKnown的值设置为true。

最后，第21行调用成员函数finishWrite来调整数据缓冲区mData下一个用来写入数据的位置，即调整成员变量mDataPos的值。

至此，注册Service组件所需要的进程间通信数据都封装到一个Parcel对象中了。接下来就要使用BC_TRANSACTION命令协议将它传递给Binder驱动程序。

5.8.1.2　发送和处理BC_TRANSACTION命令协议

回到BpServiceManager类的成员函数addService中，将要传递的通信数据封装到一个Parcel对象data中之后，接下来就调用内部的一个Binder代理对象的成员函数transact向Binder驱动程序发送一个BC_TRANSACTION命令协议。

在前面的5.1.1小节中介绍BC_TRANSACTION命令协议时提到，BC_TRANSACTION命令协议后面跟的通信数据使用一个结构体binder_transaction_data来描述，而BC_TRANSACTION命令协议又是通过IO控制命令BINDER_WRITE_READ发送到Binder驱动程序中的。因此，在发送BC_TRANSACTION命令

协议给Binder驱动程序之前，我们还需要将Parcel对象data的内容写入到一个binder_transaction_data结构体中，然后将这个binder_transaction_data结构体写入到一个binder_write_read结构体中，最后才能够通过IO控制命令BINDER_WRITE_READ将一个BC_TRANSACTION命令协议发送给Binder驱动程序处理。

Binder代理对象的成员函数transact的实现如下所示。

frameworks/base/libs/binder/BpBinder.cpp

```
01 status_t BpBinder::transact(
02     uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
03 {
04     // Once a binder has died, it will never come back to life.
05     if (mAlive) {
06         status_t status = IPCThreadState::self()->transact(
07             mHandle, code, data, reply, flags);
08         if (status == DEAD_OBJECT) mAlive = 0;
09         return status;
10     }
11
12     return DEAD_OBJECT;
13 }
```

从前面的调用过程可以知道，第一个参数code的值为ADD_SERVICE_TRANSACTION；第二个参数data包含了要传递给Binder驱动程序的进程间通信数据；第三个参数reply是一个输出参数，用来保存进程间通信结果；第四个参数flags用来描述这是一个同步的进程间通信请求，还是一个异步的进程间通信请求，它是一个默认参数，默认值为0，表示这是一个同步的进程间通信请求。

Binder代理对象内部的成员变量mAlive用来描述该Binder代理对象所引用的Binder本地对象是否还活着。如果还活着，即它的值等于1，那么第6行和第7行才会调用当前线程的IPCThreadState对象的成员函数transact向Binder驱动程序发送一个BC_TRANSACTION命令协议；否则，第12行就直接返回一个DEAD_OBJECT给调用者，表示相应的Binder本地对象已经死亡了。

Binder代理对象内部的成员变量mHandle用来描述该Binder代理对象的句柄值。由于目前正在处理的Binder代理对象指向的是一个Service Manager代理对象，因此，它的成员变量mHandle的值就等于0。

IPCThreadState类的成员函数transact的实现如下所示。

frameworks/base/libs/binder/IPCThreadState.cpp

```
01 status_t IPCThreadState::transact(int32_t handle,
02                                   uint32_t code, const Parcel& data,
03                                   Parcel* reply, uint32_t flags)
04 {
05     status_t err = data.errorCheck();
06
07     flags |= TF_ACCEPT_FDS;
08
09     ......
10
11     if (err == NO_ERROR) {
12         ......
13         err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
14     }
15
16     ......
17
18     if ((flags & TF_ONE_WAY) == 0) {
19         ......
20         if (reply) {
21             err = waitForResponse(reply);
22         } else {
23             ......
```

```
24          }
25          ......
26      } else {
27          ......
28      }
29
30      return err;
31  }
```

第5行检查Parcel对象data中的进程间通信数据是否正确，第7行将参数flags的TF_ACCEPT_FDS位设置为1，表示允许Server进程在返回结果中携带文件描述符。

如果Parcel对象data中的进程间通信数据没有问题，那么第13行就会调用成员函数writeTransactionData将它的内容写入到一个binder_transaction_data结构体中。接着第18行判断参数flags的TF_ONE_WAY位是否等于0。如果是，那么就说明这是一个同步的进程间通信请求，这时候如果用来保存通信结果的Parcel对象reply不等于NULL，那么第21行就调用成员函数waitForResponse向Binder驱动程序发送一个BC_TRANSACTION命令协议。

假设第11行、第18行和第20行的if语句均为true，接下来我们就分别分析IPCThreadState类的成员函数writeTransactionData和waitForResponse的实现。

IPCThreadState类的成员函数writeTransactionData的实现如下所示。

frameworks/base/libs/binder/IPCThreadState.cpp

```
01 status_t IPCThreadState::writeTransactionData(int32_t cmd, uint32_t binderFlags,
02     int32_t handle, uint32_t code, const Parcel& data, status_t* statusBuffer)
03 {
04     binder_transaction_data tr;
05
06     tr.target.handle = handle;
07     tr.code = code;
08     tr.flags = binderFlags;
09
10     const status_t err = data.errorCheck();
11     if (err == NO_ERROR) {
12         tr.data_size = data.ipcDataSize();
13         tr.data.ptr.buffer = data.ipcData();
14         tr.offsets_size = data.ipcObjectsCount()*sizeof(size_t);
15         tr.data.ptr.offsets = data.ipcObjects();
16     } else if (statusBuffer) {
17         ......
18     } else {
19         ......
20     }
21
22     mOut.writeInt32(cmd);
23     mOut.write(&tr, sizeof(tr));
24
25     return NO_ERROR;
26 }
```

第4行定义了一个binder_transaction_data结构体tr，接着第6行到第8行代码分别用来初始化该结构体的成员变量target.handle、code和flags。从前面的调用过程可以知道，这三个成员变量分别被初始化为0、ADD_SERVICE_TRANSACTION和TF_ACCEPT_FDS。

第10行再次确认Parcel 对象data中的进程间通信数据的正确性。如果没有问题，那么第12行到第15行代码就将它内部的数据缓冲区和偏移数组设置为binder_transaction_data结构体tr的数据缓冲区和偏移数组。

最后，第22行和第23行代码将命令协议号cmd，即BC_TRANSACTION，以及binder_transaction_data结构体tr写入到IPCThreadState类的成员变量mOut所描述的一个命令协议缓冲区中，表示它有一个BC_TRANSACTION命令协议需要发送给Binder驱动程序处理。

我们通过图5-23来描述此时命令协议缓冲区mOut的内存布局。

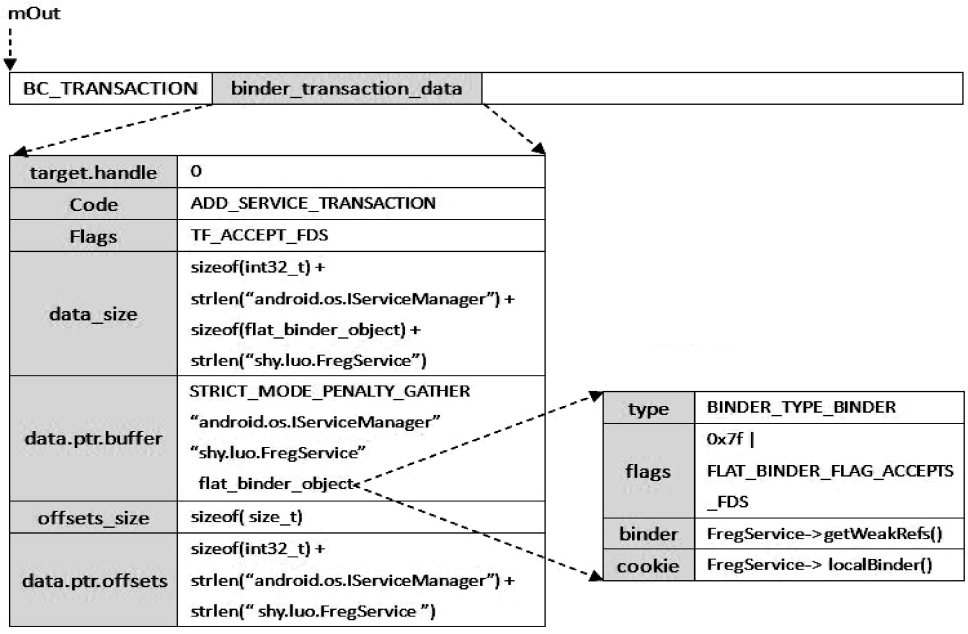


图5-23　命令协议缓冲区mOut的内存布局

将BC_TRANSACTION命令协议及其内容写入到IPCThreadState类内部的命令协议缓冲区之后，回到IPCThreadState类的成员函数transact中，接下来就继续调用成员函数waitForResponse向Binder驱动程序发送这个BC_TRANSACTION命令协议。

IPCThreadState类的成员函数waitForResponse的实现如下所示。

frameworks/base/libs/binder/IPCThreadState.cpp

```
01 status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
02 {
03     int32_t cmd;
04     int32_t err;
05
06     while (1) {
07         if ((err=talkWithDriver()) < NO_ERROR) break;
08         ......
09     }
10
11     ......
12     return err;
13 }
```

这个函数通过一个while循环不断地调用成员函数talkWithDriver来与Binder驱动程序进行交互，以便可以将前面准备好的BC_TRANSACTION命令协议发送给Binder驱动程序处理，并等待Binder驱动程序将进程间通信结果返回来。

IPCThreadState类的成员函数talkWithDriver的实现如下所示。

frameworks/base/libs/binder/IPCThreadState.cpp

```
01 status_t IPCThreadState::talkWithDriver(bool doReceive)
02 {
03     ......
```

```
04
05      binder_write_read bwr;
06
07      // Is the read buffer empty?
08      const bool needRead = mIn.dataPosition() >= mIn.dataSize();
09
10      // We don't want to write anything if we are still reading
11      // from data left in the input buffer and the caller
12      // has requested to read the next data.
13      const size_t outAvail = (!doReceive || needRead) ? mOut.dataSize() : 0;
14
15      bwr.write_size = outAvail;
16      bwr.write_buffer = (long unsigned int)mOut.data();
17
18      // This is what we'll read.
19      if (doReceive && needRead) {
20          bwr.read_size = mIn.dataCapacity();
21          bwr.read_buffer = (long unsigned int)mIn.data();
22      } else {
23          bwr.read_size = 0;
24      }
25
26      ......
27
28      // Return immediately if there is nothing to do.
29      if ((bwr.write_size == 0) && (bwr.read_size == 0)) return NO_ERROR;
30
31      bwr.write_consumed = 0;
32      bwr.read_consumed = 0;
33      status_t err;
34
35      do {
36          ......
37          if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
38              err = NO_ERROR;
39          else
40              err = -errno;
41          ......
42      } while (err == -EINTR);
43
44      ......
45
46      if (err >= NO_ERROR) {
47          if (bwr.write_consumed > 0) {
48              if (bwr.write_consumed < (ssize_t)mOut.dataSize())
49                  mOut.remove(0, bwr.write_consumed);
50              else
51                  mOut.setDataSize(0);
52          }
53          if (bwr.read_consumed > 0) {
54              mIn.setDataSize(bwr.read_consumed);
55              mIn.setDataPosition(0);
56          }
57          ......
58          return NO_ERROR;
59      }
60
61      return err;
62  }
```

在IPCThreadState类内部，除了使用缓冲区mOut来保存即将要发送给Binder驱动程序的命令协议之外，还使用缓冲区mIn来保存那些从Binder驱动程序接收到的返回协议。从前面的调用过程可以知道，缓冲区mOut里面存在一个BC_TRANSACTION命令协议。

IPCThreadState类的成员函数talkWithDriver是使用IO控制命令BINDER_WRITE_READ来与Binder驱动程序交互的，因此，它需要定义一个binder_write_read结构体来指定输入缓冲区和输出缓冲区。其中，输出缓冲区保存的是进程发送给Binder驱动程序的命令协议，而输入缓冲区保存的是Binder驱动程序发送给进程的返回协议，它们分别与IPCThreadState类内部的命令协议缓冲区mOut和返回协议缓冲区mIn对应。

参数doReceive用来描述调用者是否希望IPCThreadState类的成员函数talkWithDriver只接收Binder驱动程序发送给该进程的返回协议，它是一个默认参数，默认值为true。在前面的5.6小节中介绍Service Manager的启动过程时提到，一个进程使用IO控制命令BINDER_WRITE_READ进入到Binder驱动程序，当传递的binder_write_read结构体的输出缓冲区和输入缓冲区的长度分别等于0和大于0时，Binder驱动程序就不会处理进程给它发送的命令协议，而只会向该进程发送返回协议，这样进程就达到了只接收返回协议的效果。

IPCThreadState类的成员函数talkWithDriver是否只接收Binder驱动程序发送给它的返回协议还受到返回协议缓冲区mIn的影响。如果上次Binder发送给进程的返回协议已经处理完成，即返回协议缓冲区mIn中的返回协议已经处理完成，那么即使参数doReceive的值为true，IPCThreadState类的成员函数talkWithDriver也会向Binder驱动程序发送命令协议。因此，如果返回协议缓冲区mIn的返回协议已经处理完成，即第8行得到的变量needRead的值为true，或者调用者不是只希望接收Binder驱动程序发送给进程的返回协议，即参数doReceive的值为false，那么第13行就将变量outAvail的值设置为IPCThreadState类内部的命令协议缓冲区的长度；否则，就将它的值设置为0。接着第15行将变量outAvail的值设置为binder_write_read结构体bwr的输出缓冲区的长度write_size，并且第16行将IPCThreadState类内部的命令协议缓冲区mOut设置为binder_write_read结构体bwr的输出缓冲区write_buffer。

注意

如果变量outAvail的值等于0，那么将IPCThreadState类内部的命令协议缓冲区mOut设置为binder_write_read结构体bwr的输出缓冲区write_buffer是起不到任何效果的。

如果调用者希望接收Binder驱动程序发送给进程的返回协议，即参数doReceive的值为true，并且进程已经处理完成Binder驱动程序发送给它的返回协议，即变量needRead的值也为true，那么第20行就将IPCThreadState类内部的返回协议缓冲区mIn的长度设置为binder_write_read结构体bwr的输入缓冲区的长度read_size，并且第21行将IPCThreadState类内部的返回协议缓冲区mIn设置为binder_write_read结构体bwr的输入缓冲区read_buffer；否则，第23行就将binder_write_read结构体bwr的输入缓冲区的长度read_size设置为0，表示不要从Binder驱动程序中读取返回协议。从前面的调用过程可以知道，参数doReceive的值为默认值，即为true。假设此时IPCThreadState类内部的返回协议缓冲区mIn没有包含未处理的返回协议，那么我们得到的binder_write_read结构体bwr的输入缓冲区的长度read_size就会大于0，并且它指向了IPCThreadState类内部的返回协议缓冲区mIn。

第29行的if语句判断binder_write_read结构体bwr的输出缓冲区的长度write_size和输入缓冲区的长度read_size的值是否都等于0。如果是，就说明IPCThreadState类的成员函数talkWithDriver根本就不需要进入到Binder驱动程序，因此，函数就从第29行返回了。从前面的调用过程可以知道，binder_write_read结构体bwr的输出缓冲区write_buffer中包含了一个BC_TRANSACTION命令协议，因此，它的长度write_size大于0，函数继续往下执行。

第35行到第42行的while循环使用IO控制命令BINDER_WRITE_READ来与Binder驱动程序进行交互。由于这时候传递给Binder驱动程序的binder_write_read结构体bwr的输出缓冲区的长度write_size和输入缓冲区的长度read_size均大于0，因此，Binder驱动程序在处理IO控制命令BINDER_WRITE_READ

时，就会首先调用函数binder_thread_write来处理进程给它发送的BC_TRANSACTION命令协议，接着又会调用函数binder_thread_read来读取Binder驱动程序给进程发送的返回协议。

IPCThreadState类的成员函数talkWithDriver从Binder驱动程序中返回来之后，第47行到第52行代码将Binder驱动程序已经处理的命令协议从IPCThreadState类内部的命令协议缓冲区mOut中移除，接着第53行到第56行代码将从Binder驱动程序中读取出来的返回协议保存在IPCThreadState类内部的返回协议缓冲区mIn中。这样，IPCThreadState类的成员函数talkWithDriver返回到前面调用了它的成员函数waitForResponse时，后者就可以通过解析返回协议缓冲区mIn的内容来执行相应的操作了。

接下来，我们就进入到Binder驱动程序中，分析它的函数binder_thread_write和binder_thread_read的实现。

Binder驱动程序中的函数binder_thread_write的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 int
02 binder_thread_write(struct binder_proc *proc, struct binder_thread *thread,
03                     void __user *buffer, int size, signed long *consumed)
04 {
05         uint32_t cmd;
06         void __user *ptr = buffer + *consumed;
07         void __user *end = buffer + size;
08
09         while (ptr < end && thread->return_error == BR_OK) {
10                 if (get_user(cmd, (uint32_t __user *)ptr))
11                         return -EFAULT;
12                 ptr += sizeof(uint32_t);
13                 ......
14                 switch (cmd) {
15                 ......
16                 case BC_TRANSACTION:
17                 case BC_REPLY: {
18                         struct binder_transaction_data tr;
19
20                         if (copy_from_user(&tr, ptr, sizeof(tr)))
21                                 return -EFAULT;
22                         ptr += sizeof(tr);
23                         binder_transaction(proc, thread, &tr, cmd == BC_REPLY);
24                         break;
25                 }
26                 ......
27                 }
28
29                 *consumed = ptr - buffer;
30         }
31         return 0;
32 }
```

参数buffer指向进程传递给Binder驱动程序的一个binder_read_write结构体的输出缓冲区write_buffer。从前面的调用过程可以可以知道，这个输出缓冲区包含了一个BC_TRANSACTION命令协议，因此，第20行接着就将BC_TRANSACTION命令协议后面所跟的进程间通信数据读取出来，并且保存在binder_transaction_data结构体tr中。最后，第23行调用函数binder_transaction来处理进程发送给它的BC_TRANSACTION命令协议。

从这里可以看出，命令协议BC_REPLY的处理流程与命令协议BC_TRANSACTION类似，因此，为了方便描述，在本章接下来的内容中，当进程向Binder驱动程序发送一个BC_TRANSACTION或者BC_REPLY命令协议时，我们就跳过函数binder_thread_write的执行过程，而直接分析函数binder_transaction是如何处理这两个命令协议的。

函数binder_transaction负责处理命令协议BC_TRANSACTION和BC_REPLY，它的实现比较长，我们分段来阅读。

kernel/goldfish/drivers/staging/android/binder.c

```
001 static void
002 binder_transaction(struct binder_proc *proc, struct binder_thread *thread,
003         struct binder_transaction_data *tr, int reply)
004 {
005         struct binder_transaction *t;
006         struct binder_work *tcomplete;
007         size_t *offp, *off_end;
008         struct binder_proc *target_proc;
009         struct binder_thread *target_thread = NULL;
010         struct binder_node *target_node = NULL;
011         struct list_head *target_list;
012         wait_queue_head_t *target_wait;
013         struct binder_transaction *in_reply_to = NULL;
014         ......
015
016         if (reply) {
017                 ......
018         } else {
019                 if (tr->target.handle) {
020                         struct binder_ref *ref;
021                         ref = binder_get_ref(proc, tr->target.handle);
022                         ......
023                         target_node = ref->node;
024                 } else {
025                         target_node = binder_context_mgr_node;
026                         ......
027                 }
028                 ......
029                 target_proc = target_node->proc;
030                 ......
031                 if (!(tr->flags & TF_ONE_WAY) && thread->transaction_stack) {
032                         struct binder_transaction *tmp;
033                         tmp = thread->transaction_stack;
034                         ......
035                         while (tmp) {
036                                 if (tmp->from && tmp->from->proc == target_proc)
037                                         target_thread = tmp->from;
038                                 tmp = tmp->from_parent;
039                         }
040                 }
041         }
042         if (target_thread) {
043                 ......
044                 target_list = &target_thread->todo;
045                 target_wait = &target_thread->wait;
046         } else {
047                 target_list = &target_proc->todo;
048                 target_wait = &target_proc->wait;
049         }
```

参数reply用来描述函数binder_transaction当前要处理的是一个BC_TRANSACTION命令协议，还是一个BC_REPLY命令协议。当它的值等于0时，就表示处理的是BC_TRANSACTION命令协议；否则，就表示处理的是BC_REPLY命令协议。从前面的调用过程可以知道，参数reply的值等于0，因此，我们重点分析函数binder_transaction处理BC_TRANSACTION命令协议的过程。

我们首先回顾一下，发出该BC_TRANSACTION命令协议的源进程和源线程分别是FregServer应用程序进程及其主线程，它们分别使用binder_proc结构体proc和binder_thread 结构体thread来描述。源进

程proc发送BC_TRANSACTION命令协议给Binder驱动程序的目的是要将一个Service组件FregService注册到Service Manager中，因此，在binder_transaction_data结构体tr中，它指向的目标Binder对象是一个Binder引用对象，并且它的句柄值等于0。

由于目标Binder引用对象的句柄值等于0，即第19行的if语句为false，因此，第25行就将目标Binder实体对象target_node指向一个引用了Service Manager的Binder实体对象binder_context_mgr_node；否则，第21行就首先需要调用函数binder_get_ref来获得与句柄值tr->target.handle对应的Binder引用对象，接着第23行再通过这个Binder引用对象的成员变量node来找到目标Binder实体对象target_node。

注意

引用了Service Manager的Binder实体对象binder_context_mgr_node是Binder驱动程序在Service Manager启动时创建的，具体可以参考5.6.2小节的内容。

找到了目标Binder实体对象之后，第29行就可以根据它的成员变量proc来找到目标进程target_proc。从理论上说，找到了目标进程target_proc之后，Binder驱动程序就可以向它发送一个BR_TRANSACTION返回协议，以便它可以处理注册Service组件FregService的进程间通信请求。发送给目标进程target_proc的BR_TRANSACTION返回协议最终是由它的空闲Binder线程来处理的。这些空闲的Binder线程可以划分为两种类型：第一种是因为无事可做而空闲；第二种不是真的空闲，而是它在处理某个事务的过程中，需要等待其他线程来完成另外一个事务。如果Binder驱动程序能够从目标进程target_proc中挑选出一个属于第二种类型的空闲Binder线程来处理BR_TRANSACTION返回协议，并且又不会影响该线程处理它原来的事务，那么Binder驱动程序就可以充分地利用目标进程target_proc的空闲Binder线程来处理进程间通信请求了。

考虑这样一个情景。进程S中有两个Binder线程A和B，其中，线程B处于空闲状态，而线程A正在处理事务T1。线程A在处理事务T1时，需要请求进程T中的线程C来完成事务T2，这时候线程A也处于空闲状态，因为它要等待线程C完成事务T2。线程C在处理事务T2时，又需要请求进程S来完成事务T3。这时候Binder驱动程序将进程S的事务T3是分配给线程A还是线程B来处理呢？最优的方案是分配给线程A来处理，因为即使Binder驱动程序将事务T3分配给线程B来处理，线程A仍然不可以去处理其他事务，它需要等待线程C处理完成事务T2，以及线程B处理完成事务T3，然后再继续处理事务T1。

第31行到第40行代码做了一个优化，它尝试在目标进程target_proc中找到一个属于第二种类型的Binder空闲线程target_thread来处理一个BR_TRANSACTION返回协议，以便可以提高目标进程target_proc的进程间通信并发处理能力。在前面的5.1.1小节中介绍结构体binder_transaction时已经详细分析过这个优化方案了，因此，读者可以结合这段代码来重新理解结构体binder_transaction的定义。不过有一个地方需要注意，只有当前正在处理的进程间通信请求是同步的，即binder_transaction结构体tr的成员变量flags的 TF_ONE_WAY位等于0，才会执行该优化方案，因为异步的进程间通信请求根本就不会涉及一个线程等待另外一个线程的问题。

如果Binder驱动程序在目标进程target_proc中找到了一个最优的目标线程target_thread来接收BR_TRANSACTION返回协议，即第42行的if语句为true，那么就44行和第45行就将变量target_list和target_wait分别指向该目标线程target_thread的todo队列和wait等待队列；否则，第47行和第48行就将变量target_list和target_wait分别指向该目标进程target_proc的todo队列和wait等待队列。有了目标todo队列target_list和目标wait等待队列target_wait之后，函数接下来就可以将一个与BR_TRANSACTION返回协议相关的待处理工作项加入到目标todo队列target_list中，以及通过目标wait等待队列target_wait将目标进程或者目标线程唤醒来处理这个工作项。

函数binder_transaction继续往下执行。

kernel/goldfish/drivers/staging/android/binder.c

```
050         t = kzalloc(sizeof(*t), GFP_KERNEL);
051         ......
052
053         tcomplete = kzalloc(sizeof(*tcomplete), GFP_KERNEL);
054         ......
055
056         if (!reply && !(tr->flags & TF_ONE_WAY))
057                 t->from = thread;
058         else
059                 t->from = NULL;
060         t->sender_euid = proc->tsk->cred->euid;
061         t->to_proc = target_proc;
062         t->to_thread = target_thread;
063         t->code = tr->code;
064         t->flags = tr->flags;
065         t->priority = task_nice(current);
066         t->buffer = binder_alloc_buf(target_proc, tr->data_size,
067                 tr->offsets_size, !reply && (t->flags & TF_ONE_WAY));
068         ......
069         t->buffer->allow_user_free = 0;
070         ......
071         t->buffer->transaction = t;
072         t->buffer->target_node = target_node;
073         if (target_node)
074                 binder_inc_node(target_node, 1, 0, NULL);
075
076         offp = (size_t *)(t->buffer->data + ALIGN(tr->data_size, sizeof(void *)));
077
078         if (copy_from_user(t->buffer->data, tr->data.ptr.buffer, tr->data_size)) {
079                 ......
080                 goto err_copy_data_failed;
081         }
082         if (copy_from_user(offp, tr->data.ptr.offsets, tr->offsets_size)) {
083                 ......
084                 goto err_copy_data_failed;
085         }
086         ......
087
088         off_end = (void *)offp + tr->offsets_size;
```

第50行分配了一个binder_transaction结构体t，后面会将它封装为一个BINDER_WORK_TRANSACTION类型的工作项加入到目标todo队列target_list中，以便目标线程可以接收到一个BR_TRANSACTION返回协议。

第53行分配了一个binder_work结构体tcomplete，后面会将它封装成一个BINDER_WORK_TRANSACTION_COMPLETE类型的工作项加入到源线程thread的todo队列中，以便该线程知道可以马上返回用户空间，并且知道它之前给Binder驱动程序发送的BC_TRANSACTION命令协议已经被接收了。

第56行到第72行代码初始化前面分配的binder_transaction结构体t。如果函数正在处理的是一个BC_TRANSACTION命令协议，并且它所描述的是一个同步的进程间通信请求，那么第57行就会将binder_transaction结构体t的成员变量from指向源线程thread，以便目标进程target_proc或者目标线程target_thread处理完该进程间通信请求之后，能够找回发出该进程间通信请求的线程，最终将进程间通信结果返回给它。第66行和第77行为binder_transaction结构体t分配一个内核缓冲区，以便可以将进程间通信数据复制到它里面，最后传递给目标进程target_proc或者目标线程target_thread处理。

注意

这个内核缓冲区是在目标进程target_proc中分配的。第74行调用函数binder_inc_node来增加目标Binder实体对象的强引用计数，因为binder_transaction结构体t通过成员变量target_node引用了它。

第78行和第82行代码分别将binder_transaction_data结构体tr的数据缓冲区，以及偏移数组的内容复制到分配给binder_transaction结构体t的内核缓冲区中。

第76行和第88行代码分别计算分配给binder_transaction结构体t的内核缓冲区中用来保存偏移数组的开始位置offp和结束位置off_end。接下来，函数就根据这两个位置来遍历进程间通信数据中的Binder对象，以便可以对它们进行处理。

函数binder_transaction继续往下执行。

kernel/goldfish/drivers/staging/android/binder.c

```
089         for (; offp < off_end; offp++) {
090                 struct flat_binder_object *fp;
091                 ......
092                 fp = (struct flat_binder_object *)(t->buffer->data + *offp);
093                 switch (fp->type) {
094                 case BINDER_TYPE_BINDER:
095                 case BINDER_TYPE_WEAK_BINDER: {
096                         struct binder_ref *ref;
097                         struct binder_node *node = binder_get_node(proc, fp->binder);
098                         if (node == NULL) {
099                                 node = binder_new_node(proc, fp->binder, fp->cookie);
100                                 ......
101                                 node->min_priority = fp->flags & FLAT_BINDER_FLAG_PRIORITY_MASK;
102                                 node->accept_fds = !!(fp->flags & FLAT_BINDER_FLAG_ACCEPTS_FDS);
103                         }
104                         ......
105                         ref = binder_get_ref_for_node(target_proc, node);
106                         ......
107                         if (fp->type == BINDER_TYPE_BINDER)
108                                 fp->type = BINDER_TYPE_HANDLE;
109                         else
110                                 fp->type = BINDER_TYPE_WEAK_HANDLE;
111                         fp->handle = ref->desc;
112                         binder_inc_ref(ref, fp->type == BINDER_TYPE_HANDLE, &thread->todo);
113                         ......
114                 } break;
115                 ......
116                 }
117
118         }
```

第89行到第118行的for循环依次处理进程间通信数据中的Binder对象。如果Binder驱动程序是第一次碰到这些 Binder对象，那么Binder驱动程序就会根据它们的类型分别创建一个Binder实体对象或者一个Binder引用对象；否则，就会将之前为它们创建的Binder实体对象或者Binder引用对象获取回来，以便可以增加它们的引用计数，避免它们过早地被销毁。

从前面的调用过程可以知道，进程间通信数据中包含有一个类型为BINDER_TYPE_BINDER的Binder对象，即一个flat_binder_object结构体，它用来描述即将要注册到Service Manager中的Service组件FregService。由于Binder驱动程序是第一次碰到这个Binder对象，因此，第97行调用函数binder_get_node就无法获得一个引用了它的Binder实体对象，接着第99行就会调用函数binder_new_node为它创建一个Binder实体对象node。在创建Binder实体对象node时，第101行和第102行代码就会根据从用户空间传递进来的flat_binder_object结构体的内容来设置它的最小线程优先级min_priority，以及是否接收文件描述符标志accept_fds。

在前面的5.6小节中介绍Service Manager的启动过程时，我们已经分析过函数binder_new_node的实现了，下面就简要地分析函数binder_get_node的实现，它的定义如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static struct binder_node *
02 binder_get_node(struct binder_proc *proc, void __user *ptr)
03 {
04         struct rb_node *n = proc->nodes.rb_node;
05         struct binder_node *node;
06 
07         while (n) {
08                 node = rb_entry(n, struct binder_node, rb_node);
09 
10                 if (ptr < node->ptr)
11                         n = n->rb_left;
12                 else if (ptr > node->ptr)
13                         n = n->rb_right;
14                 else
15                         return node;
16         }
17         return NULL;
18 }
```

函数binder_get_node根据一个用户空间地址ptr在目标进程proc中找到一个对应的Binder实体对象。在前面的5.1.1小节中介绍结构体binder_proc时提到，一个进程中的所有Binder实体对象都以它们的成员变量ptr作为关键字保存在进程内部的一个红黑树nodes中。因此，函数就在目标进程proc的Binder实体对象红黑树nodes中检查是否存在一个与参数ptr对应的Binder实体对象。如果存在，就将对应的Binder对象返回给调用者；否则，就返回一个NULL值给调用者。

回到函数binder_transaction中，因为函数接下来就要将即将要注册的Service组件FregService从源进程proc传递到目标进程target_proc中，因此，第105行就需要调用函数binder_get_ref_for_node在目标进程target_proc中创建一个Binder引用对象来引用该Service组件FregService。

函数binder_get_ref_for_node的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static struct binder_ref *
02 binder_get_ref_for_node(struct binder_proc *proc, struct binder_node *node)
03 {
04         struct rb_node *n;
05         struct rb_node **p = &proc->refs_by_node.rb_node;
06         struct rb_node *parent = NULL;
07         struct binder_ref *ref, *new_ref;
08 
09         while (*p) {
10                 parent = *p;
11                 ref = rb_entry(parent, struct binder_ref, rb_node_node);
12 
13                 if (node < ref->node)
14                         p = &(*p)->rb_left;
15                 else if (node > ref->node)
16                         p = &(*p)->rb_right;
17                 else
18                         return ref;
19         }
20         new_ref = kzalloc(sizeof(*ref), GFP_KERNEL);
21         ......
22         new_ref->proc = proc;
23         new_ref->node = node;
24         rb_link_node(&new_ref->rb_node_node, parent, p);
25         rb_insert_color(&new_ref->rb_node_node, &proc->refs_by_node);
26 
27         new_ref->desc = (node == binder_context_mgr_node) ? 0 : 1;
```

```
28          for (n = rb_first(&proc->refs_by_desc); n != NULL; n = rb_next(n)) {
29                  ref = rb_entry(n, struct binder_ref, rb_node_desc);
30                  if (ref->desc > new_ref->desc)
31                          break;
32                  new_ref->desc = ref->desc + 1;
33          }
34
35          p = &proc->refs_by_desc.rb_node;
36          while (*p) {
37                  parent = *p;
38                  ref = rb_entry(parent, struct binder_ref, rb_node_desc);
39
40                  if (new_ref->desc < ref->desc)
41                          p = &(*p)->rb_left;
42                  else if (new_ref->desc > ref->desc)
43                          p = &(*p)->rb_right;
44                  else
45                          BUG();
46          }
47          rb_link_node(&new_ref->rb_node_desc, parent, p);
48          rb_insert_color(&new_ref->rb_node_desc, &proc->refs_by_desc);
49          if (node) {
50                  hlist_add_head(&new_ref->node_entry, &node->refs);
51                  ......
52          } else {
53                  ......
54          }
55          return new_ref;
56  }
```

函数binder_get_ref_for_node首先判断是否已经在目标进程proc中为Binder实体对象node创建过一个Binder引用对象。如果已经创建过，那么就会将对应的Binder引用对象返回给调用者；否则，就会首先创建一个Binder引用对象来引用该Binder实体对象node，然后再将它返回给调用者。

在前面的5.1.1小节中介绍结构体binder_proc时提到，一个进程中的所有Binder引用对象都以它们的成员变量node作为关键字保存在一个红黑树refs_by_node中。因此，第9行到第19行的while循环就在目标进程proc的红黑树refs_by_node中检查是否已经存在一个与Binder实体对象node对应的Binder引用对象。如果存在，就直接将它返回给调用者；否则，接下来第20行代码就会在目标进程proc中为Binder实体对象node创建一个Binder引用对象new_ref，并且第24行和第25行代码将它添加到目标进程proc的红黑树refs_by_node中。

第27行到第33行代码为新创建的Binder引用对象new_ref分配句柄值。第27行检查Binder实体对象node是否引用了Service Manager的Binder实体对象binder_context_mgr_node。如果是，那么就将Binder引用对象new_ref的句柄值设置为0；否则，就先将Binder引用对象new_ref的句柄值设置为1，接着第28行到第33行的for循环再根据实际情况对它进行调整。第28行到第33行的for循环实际上是在目标进程proc中找到一个未使用的最小的句柄值来作为新创建的Binder引用对象new_ref的句柄值。

注意

一个进程中的Binder引用对象除了保存在红黑树refs_by_node中之外，还以它们的句柄值作为关键字保存在另外一个红黑树refs_by_desc中。当通过函数rb_first和rb_next遍历红黑树refs_by_desc时，就会按照句柄值从小到大的顺序来获得这些Binder引用对象，因此，第28行到第33行的for循环在目标进程proc中找到的未使用的句柄值是最小的。

为新创建的Binder引用对象new_ref分配好句柄值之后，第36行到第46行的while循环再次确认前面为Binder引用对象new_ref分配的句柄值是有效的。接着第47行和第48行代码将Binder引用对象new_ref

添加到目标进程proc的红黑树refs_by_desc中。最后，第50行将Binder引用对象new_ref添加到它所引用的Binder实体对象node的Binder引用对象列表中。

回到函数binder_transaction中，第107行到第111行代码将flat_binder_object结构体fp的类型修改为BINDER_TYPE_HANDLE，并且设置好它的句柄值。这是因为当Binder驱动程序将进程间通信数据传递给目标进程target_proc时，进程间通信数据中的Binder实体对象就变成了Binder引用对象，因此，就需要修改flat_binder_object结构体fp的类型。

在目标进程target_proc创建好一个Binder引用对象ref之后，接着就要将它传递给该目标进程了。在传递的过程中，必须要保证Binder引用对象ref不会被销毁，因此，第112行就调用函数binder_inc_ref来增加它的引用计数。

注意

前面在创建Binder引用对象ref时，尚未增加过与它所引用的Binder实体对象对应的Binder本地对象的引用计数，因此，在调用函数binder_inc_ref来增加Binder引用对象ref的引用计数时，需要将源线程thread的todo队列作为第三个参数传进去，以便Binder驱动程序可以将一个类型为BINDER_WORK_NODE的工作项添加到它里面。这样，当源线程thread从Binder驱动程序返回到用户空间时，就可以增加相应的Binder本地对象，即Service组件FregService的引用计数了。有关Binder对象引用计数的内容，可以参考前面5.2小节的内容。

函数binder_transaction继续往下执行。

kernel/goldfish/drivers/staging/android/binder.c

```
119         if (reply) {
120                 ......
121         } else if (!(t->flags & TF_ONE_WAY)) {
122                 BUG_ON(t->buffer->async_transaction != 0);
123                 t->need_reply = 1;
124                 t->from_parent = thread->transaction_stack;
125                 thread->transaction_stack = t;
126         } else {
127                 BUG_ON(target_node == NULL);
128                 BUG_ON(t->buffer->async_transaction != 1);
129                 if (target_node->has_async_transaction) {
130                         target_list = &target_node->async_todo;
131                         target_wait = NULL;
132                 } else
133                         target_node->has_async_transaction = 1;
134         }
135         t->work.type = BINDER_WORK_TRANSACTION;
136         list_add_tail(&t->work.entry, target_list);
137         tcomplete->type = BINDER_WORK_TRANSACTION_COMPLETE;
138         list_add_tail(&tcomplete->entry, &thread->todo);
139         if (target_wait)
140                 wake_up_interruptible(target_wait);
141         return;
142
143         ......
144 }
```

如果函数binder_transaction正在处理的是一个同步的进程间通信请求，即binder_transaction结构体t的成员变量flags的TF_ONE_WAY位等于0，那么第123行就设置它的成员变量need_reply的值为1，表示它需要等待回复。接着第124行和第125行将事务t压入到源线程thread的事务堆栈transaction_stack中。

如果函数binder_transaction正在处理的是一个异步的进程间通信请求，即binder_transaction结构体t的成员变量flags的TF_ONE_WAY位等于1，那么第129行的if语句就会检查目标Binder实体对象target_node当

前是否正在处理异步事务。如果是，那么它的成员变量has_async_transaction的值就会等于1。在这种情况下，Binder驱动程序就需要将binder_transaction结构体t封装成一个工作项添加到目标Binder实体对象target_node的async_todo队列中去等待处理，而不应该放到目标进程target_proc或者目标线程target_thread的todo队列中去等待处理。因此，第130行和第131行就需要修改目标todo队列target_list和目标wait等待队列target_wait的值。在前面的5.1.1小节中介绍结构体binder_node时提到，一个Binder实体对象的所有异步事务都必须是串行执行的，这是为了保证它所在的进程能够优先处理同步事务。因此，当一个Binder实体对象正在处理一个异步事务时，接下来的其余异步事务就必须要添加到它的异步事务队列async_todo中去等待处理。如果目标Binder实体对象target_node当前不是正在处理异步事务，那么就可以将binder_transaction结构体t封装成一个工作项添加到目标进程target_proc或者目标线程target_thread的todo队列中去等待处理，但是要将它的成员变量has_async_transaction的值设置为1。

第135行和第136行将binder_transaction结构体t封装成一个类型为BINDER_WORK_TRANSACTION的工作项添加到目标进程target_proc或者目标线程target_thread的todo队列中，并且第140行将目标进程target_proc或者目标线程target_thread唤醒，以便它们可以处理这个工作项。如果目标todo队列target_list指向的是一个Binder实体对象的异步事务队列async_todo，那么目标wait等待队列target_wait就会等于NULL，这时候就不需要将目标进程target_proc或者目标线程target_thread唤醒了

第137行和第138行将binder_work结构体tcomplete封装成一个类型为BINDER_WORK_TRANSACTION_COMPLETE的工作项添加到源线程thread的todo队列中，以便它从Binder驱动程序返回到用户空间之前，可以处理该工作项。

程序执行到这里时，源线程thread、目标进程target_proc或者目标线程target_thread就会并发地去处理各自的todo队列中的工作项了。为了方便描述，我们假设源线程thread首先处理它的todo队列中类型为BINDER_WORK_TRANSACTION_COMPLETE的工作项，接着目标进程target_proc或者目标线程target_thread再处理它们的todo队列中类型为BINDER_WORK_TRANSACTION的工作项。

源线程thread从函数binder_transaction返回到函数binder_thread_write之后，再返回到函数binder_ioctl中。由于从用户空间传进来的binder_write_read结构体的输入缓冲区的长度read_size大于0，因此，Binder驱动程序接下来就会调用函数binder_thread_read来处理源线程thread的todo队列中类型为BINDER_WORK_TRANSACTION_COMPLETE的工作项。

函数binder_thread_read的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static int
02 binder_thread_read(struct binder_proc *proc, struct binder_thread *thread,
03         void  __user *buffer, int size, signed long *consumed, int non_block)
04 {
05         void __user *ptr = buffer + *consumed;
06         void __user *end = buffer + size;
07
08         int ret = 0;
09         ......
10
11         while (1) {
12                 uint32_t cmd;
13                 ......
14                 struct binder_work *w;
15                 ......
16
17                 if (!list_empty(&thread->todo))
18                         w = list_first_entry(&thread->todo, struct binder_work, entry);
19                 else if (!list_empty(&proc->todo) && wait_for_proc_work)
20                         w = list_first_entry(&proc->todo, struct binder_work, entry);
```

```
21                  else {
22                          if (ptr - buffer == 4 && !(thread->looper & BINDER_LOOPER_STATE_NEED_RETURN))
/* no data added */
23                                  goto retry;
24                          break;
25                  }
26
27                  ......
28
29                  switch (w->type) {
30                  ......
31                  case BINDER_WORK_TRANSACTION_COMPLETE: {
32                          cmd = BR_TRANSACTION_COMPLETE;
33                          if (put_user(cmd, (uint32_t __user *)ptr))
34                                  return -EFAULT;
35                          ptr += sizeof(uint32_t);
36                          ......
37
38                          list_del(&w->entry);
39                          kfree(w);
40                          ......
41                  } break;
42                  ......
43                  }
44          }
45
46 done:
47          *consumed = ptr - buffer;
48          ......
49          return 0;
50 }
```

第18行将线程thread的todo队列中类型为BINDER_WORK_TRANSACTION_COMPLETE的工作项取出来，接着第33行就将一个BR_TRANSACTION_COMPLETE返回协议写入到用户空间提供的缓冲区中。从这里可以看出，Binder驱动程序处理类型为BINDER_WORK_TRANSACTION_COMPLETE的工作项的方式是向相应的进程发送一个BR_TRANSACTION_COMPLETE返回协议。

函数binder_thread_read执行完成之后，就返回到函数binder_ioctl中，然后再返回到IPCThreadState类的成员函数talkWithDriver中，最后又返回到IPCThreadState类的成员函数waitForResponse中，以便可以处理Binder驱动程序给进程发送的BR_TRANSACTION_COMPLETE返回协议。

IPCThreadState类的成员函数waitForResponse的实现如下所示。

frameworks/base/libs/binder/IPCThreadState.cpp

```
01 status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
02 {
03     int32_t cmd;
04     int32_t err;
05
06     while (1) {
07         if ((err=talkWithDriver()) < NO_ERROR) break;
08         err = mIn.errorCheck();
09         if (err < NO_ERROR) break;
10         if (mIn.dataAvail() == 0) continue;
11
12         cmd = mIn.readInt32();
13
14         ......
15
16         switch (cmd) {
17         case BR_TRANSACTION_COMPLETE:
18             if (!reply && !acquireResult) goto finish;
```

```
19              break;
20          ......
21          }
22      }
23
24      ......
25      return err;
26  }
```

IPCThreadState类的成员函数talkWithDriver已经将Binder驱动程序给进程发送的BR_TRANSACTION_COMPLETE返回协议保存在内部的返回协议缓冲区mIn中，因此，第12行就可以通过返回协议缓冲区mIn来获得这个BR_TRANSACTION_COMPLETE返回协议。IPCThreadState类的成员函数waitForResponse对BR_TRANSACTION_COMPLETE返回协议的处理很简单，它跳出了第16行到第21行的switch语句之后，就重新执行外层的while循环，即再次调用成员函数talkWithDriver来与Binder驱动程序交互。

再次进入到IPCThreadState类的成员函数talkWithDriver中时，由于IPCThreadState类内部的命令协议缓冲区mOut中的命令协议，以及返回协议缓冲区mIn中的返回协议都已经处理完成了，因此，当它再次通过IO控制命令BINDER_WRITE_READ进入到Binder驱动程序的函数binder_ioctl中时，就会直接调用函数binder_thread_read等待目标进程将上次发出的进程间通信请求的结果返回来。

5.8.1.3　发送和处理BR_TRANSACTION返回协议

在前面的5.6小节中分析Service Manager的启动过程时提到，Service Manager在启动完成之后，它的主线程就会进入到Binder驱动程序的函数binder_thread_read中去等待它的 todo队列或者它所属进程的todo队列中有新的工作项需要处理。

假设在前面的5.8.1.2小节中，Binder驱动程序将一个类型为BINDER_WORK_TRANSACTION的工作项添加到了Service Manager进程的todo队列中，那么接下来Service Manager的主线程就会被唤醒，继续往下执行函数binder_thread_read。

函数binder_thread_read负责处理一个线程或者一个进程的todo队列中的工作项。处理完成之后，它就会向目标进程发送一个返回协议。

函数binder_thread_read的实现比较长，我们分段来阅读。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static int
02 binder_thread_read(struct binder_proc *proc, struct binder_thread *thread,
03         void  __user *buffer, int size, signed long *consumed, int non_block)
04 {
05         void __user *ptr = buffer + *consumed;
06         void __user *end = buffer + size;
07
08         int ret = 0;
09         ......
10
11         while (1) {
12                 uint32_t cmd;
13                 struct binder_transaction_data tr;
14                 struct binder_work *w;
15                 struct binder_transaction *t = NULL;
16
17                 if (!list_empty(&thread->todo))
18                         w = list_first_entry(&thread->todo, struct binder_work, entry);
19                 else if (!list_empty(&proc->todo) && wait_for_proc_work)
20                         w = list_first_entry(&proc->todo, struct binder_work, entry);
21                 else {
22                     if (ptr - buffer == 4 && !(thread->looper & BINDER_LOOPER_STATE_NEED_RETURN))
/* no data added */
```

```
23                                  goto retry;
24                          break;
25                  }
26
27                  ......
28
29                  switch (w->type) {
30                  case BINDER_WORK_TRANSACTION: {
31                          t = container_of(w, struct binder_transaction, work);
32                  } break;
33                  ......
34                  }
```

第17行的if语句首先检查线程thread自己的todo队列中是否有工作项需要处理。如果没有，第19行的if语句再检查它所属进程proc的todo队列中是否有工作项需要处理。只要其中的一个todo队列中有工作项需要处理，函数binder_thread_read就将它取出来处理，并且保存在binder_work结构体w中。前面我们假设Binder驱动程序已经将一个类型为BINDER_WORK_TRANSACTION的工作项添加到了Service Manager进程的todo队列中，因此，第20行就会将该工作项取出来，并且保存在binder_work结构体w中。

由于binder_work结构体w的类型为BINDER_WORK_TRANSACTION，即它是一个嵌入在一个binder_transaction结构体中的工作项，因此，第31行就可以安全地将它转换为一个binder_transaction结构体t。

函数binder_thread_read继续往下执行。

kernel/goldfish/drivers/staging/android/binder.c

```
35                  if (t->buffer->target_node) {
36                          struct binder_node *target_node = t->buffer->target_node;
37                          tr.target.ptr = target_node->ptr;
38                          tr.cookie =  target_node->cookie;
39                          t->saved_priority = task_nice(current);
40                          if (t->priority < target_node->min_priority &&
41                              !(t->flags & TF_ONE_WAY))
42                                  binder_set_nice(t->priority);
43                          else if (!(t->flags & TF_ONE_WAY) ||
44                                   t->saved_priority > target_node->min_priority)
45                                  binder_set_nice(target_node->min_priority);
46                          cmd = BR_TRANSACTION;
47                  } else {
48                          ......
49                  }
50                  tr.code = t->code;
51                  tr.flags = t->flags;
52                  tr.sender_euid = t->sender_euid;
53
54                  if (t->from) {
55                          struct task_struct *sender = t->from->proc->tsk;
56                          tr.sender_pid = task_tgid_nr_ns(sender, current->nsproxy->pid_ns);
57                  } else {
58                          ......
59                  }
60
61                  tr.data_size = t->buffer->data_size;
62                  tr.offsets_size = t->buffer->offsets_size;
63                  tr.data.ptr.buffer = (void *)t->buffer->data + proc->user_buffer_offset;
64                  tr.data.ptr.offsets = tr.data.ptr.buffer + ALIGN(t->buffer->data_size, sizeof(void *));
65
66                  if (put_user(cmd, (uint32_t __user *)ptr))
67                          return -EFAULT;
68                  ptr += sizeof(uint32_t);
69                  if (copy_to_user(ptr, &tr, sizeof(tr)))
```

```
70                          return -EFAULT;
71                  ptr += sizeof(tr);
72                  ......
73
74                  list_del(&t->work.entry);
75                  t->buffer->allow_user_free = 1;
76                  if (cmd == BR_TRANSACTION && !(t->flags & TF_ONE_WAY)) {
77                          t->to_parent = thread->transaction_stack;
78                          t->to_thread = thread;
79                          thread->transaction_stack = t;
80                  } else {
81                          t->buffer->transaction = NULL;
82                          kfree(t);
83                          ......
84                  }
85                  break;
86          }
87
88          return 0;
89  }
```

Binder驱动程序处理BINDER_WORK_TRANSACTION类型的工作项的方式是向目标线程thread发送一个BR_TRANSACTION返回协议。在前面的5.1.1小节中介绍Binder返回协议BinderDriverReturnProtocol时提到，BR_TRANSACTION返回协议的内容是使用一个binder_transaction_data结构体来描述的，因此，Binder驱动程序在向目标线程thread发送一个BR_TRANSACTION返回协议之前，需要定义一个binder_transaction_data结构体tr，并且将保存在binder_transaction结构体t中的进程间通信数据复制到它里面。

从前面5.8.1.2小节的内容可以知道，binder_transaction结构体t的内核缓冲区buffer的成员变量target_node指向了引用了Service Manager的Binder实体对象binder_context_mgr_node，它的值不等于NULL。因此，第37行和第38行就将它里面的目标Binder本地对象信息复制到binder_transaction_data结构体tr中，以便目标线程thread接收到Binder驱动程序给它发送的BR_TRANSACTION返回协议之后，可以将该返回协议交给指定的Binder本地对象来处理。

在前面的5.1.1小节中介绍结构体binder_node时提到，每一个Binder实体对象在处理一个进程间通信请求时，都对它所运行在的线程的线程优先级有最低要求，这个最小线程优先级就保存在它的成员变量min_priority中。同时，Binder驱动程序正在处理的binder_transaction结构体t的成员变量priority也包含了一个线程优先级，从前面5.8.1.2小节的内容可以知道，这个线程优先级就是发出由binder_transaction结构体t所描述的进程间通信请求的源线程的线程优先级。由于Binder进程间通信的本质是使用目标线程来代替源线程执行一个操作，因此，Binder驱动程序就尽量地将目标线程模拟成源线程，使得源线程感觉不到目标线程的存在。为了达到这个目的，Binder驱动程序需要做的一件事情就是保证目标线程的线程优先级不低于源线程的线程优先级。基于上述两个特性，第40行到第45行代码就需要相应地修改目标线程thread的线程优先级。不过，在修改目标线程thread的线程优先级之前，第39行首先将它原来的线程优先级保存在binder_transaction结构体t的成员变量saved_priority中，以便它处理完成该进程间通信请求之后，Binder驱动程序可以恢复它原来的线程优先级。

如果binder_transaction结构体t描述的是一个同步的进程间通信请求，并且源线程的线程优先级t->priority高于目标Binder实体对象target_node所要求的最小线程优先级min_priority，即第40行的if语句为true，那么第42行就将目标线程thread的线程优先级设置为源线程的线程优先级。

当binder_transaction结构体t描述的是一个异步的进程间通信请求时，即第43行的if语句为true，那么Binder驱动程序在修改目标线程thread的线程优先级时，就不需要考虑源线程的线程优先级了。这是因为源线程不需要等待目标线程的进程间通信结果。因此，Binder驱动程序就不需要将目标线程模

拟成源线程来执行。不过，这时候Binder驱动程序仍然需要保证目标线程thread的线程优先级不能低于目标Binder实体对象target_node的最小线程优先级min_priority。因此，第43行的if语句为true时，需要进一步检查目标Binder实体对象target_node的最小线程优先级min_priority是否高于目标线程thread的线程优先级。如果是，那么第45行就将目标线程thread的线程优先级设置为目标Binder实体对象target_node的最小线程优先级min_priority。

第50行和第51行代码将binder_transaction结构体t中的进程间通信数据复制到binder_transaction_data结构体tr中。从前面5.8.1.2小节中的图5-23可以知道，复制到binder_transaction_data结构体tr中的成员变量code和flags的值分别为ADD_SERVICE_TRANSACTION和TF_ACCEPT_FDS。

第52行到第59行代码设置binder_transaction_data结构体tr的成员变量sender_euid和sender_pid，它们分别指向源线程的有效用户ID，以及线程组PID，这样目标线程thread在处理一个进程间通信请求时，就可以识别出源线程的身份，以便做一些安全性和合法性检查。

第61行到第64行代码将binder_transaction结构体t中的数据缓冲区和偏移数组的内容复制到binder_transaction_data结构体tr中。

注意

binder_transaction结构体t中的数据缓冲区和偏移数组保存在Binder驱动程序为目标进程分配的内核缓冲区中，它们使用内核空间地址来描述。

在前面的5.1.4小节中介绍进程内核缓冲区的分配过程时提到，Binder驱动程序为进程分配的内核缓冲区是用来传输进程间通信数据的，即将进程间通信数据从内核地址空间传输到用户地址空间。为了减少一次数据复制操作，Binder驱动程序分配给进程的内核缓冲区同时映射到了进程的内核地址空间和用户地址空间。因此，第63行和第64行代码其实并不是真的将binder_transaction结构体t中的数据缓冲区和偏移数组的内容复制到binder_transaction_data结构体tr中，而是修改binder_transaction_data结构体tr中的数据缓冲区和偏移数组的地址值，使它们指向binder_transaction结构体t中的数据缓冲区和偏移数组。由于Binder驱动程序为目标进程proc分配的内核缓冲区的内核空间地址和用户空间地址相差一个固定值，并且保存在它的成员变量user_buffer_offset中，因此，知道了binder_transaction结构体t中的数据缓冲区和偏移数组的内核空间地址之后，就很容易知道它们对应的用户空间地址了。

binder_transaction_data结构体tr的进程间通信数据设置完成之后，第66行到第71行代码将它以及它所对应的返回协议BR_TRANSACTION复制到由目标线程thread提供的一个用户空间缓冲区中。

第74行将binder_work结构体w从目标线程thread或者目标进程的todo队列中删除，因为它所描述的工作项已经得到处理了。接着第75行将binder_transaction结构体t的成员变量allow_user_free的值设置为1，表示Binder驱动程序为它所分配的内核缓冲区允许目标线程thread在用户空间中发出BC_FREE_BUFFER命令协议来释放。

如果Binder驱动程序向目标线程thread发送的是一个BR_TRANSACTION返回协议，并且binder_transaction结构体t的成员变量flags的TF_ONE_WAY位等于0，即第76行的if语句为true，那么就说明Binder驱动程序正在请求目标线程thread执行一个同步的进程间通信请求。这时候第77行到第79行就将binder_transaction结构体t压入到目标线程thread的事务堆栈transaction_stack中，以便Binder驱动程序以后可以从目标线程thread所属进程的Binder线程池中选择一个最优的空闲Binder线程来处理其他的进程间通信请求。

如果Binder驱动程序正在处理的不是一个同步的进程间通信请求，即第76行的if语句为false，那么第82行就释放binder_transaction结构体t所占用的内核空间；否则，就需要等到该同步的进程间通信请求处理完成之后，才可以释放binder_transaction结构体t所占用的内核空间。

这里所说的目标线程thread，即为Service Manager进程的主线程。从前面5.6小节的内容可以知道，当Service Manager进程的主线程在Binder驱动程序的函数binder_thread_read中被唤醒，并且返回到用户空间之后，它就会调用Service Manager进程中的函数binder_parse来处理从Binder驱动程序接收到的返回协议。

函数binder_parse的实现如下所示。

frameworks/base/cmds/servicemanager/binder.c

```
01 int binder_parse(struct binder_state *bs, struct binder_io *bio,
02                  uint32_t *ptr, uint32_t size, binder_handler func)
03 {
04     int r = 1;
05     uint32_t *end = ptr + (size / 4);
06
07     while (ptr < end) {
08         uint32_t cmd = *ptr++;
09         ......
10         switch(cmd) {
11         ......
12         case BR_TRANSACTION: {
13             struct binder_txn *txn = (void *) ptr;
14             ......
15             if (func) {
16                 unsigned rdata[256/4];
17                 struct binder_io msg;
18                 struct binder_io reply;
19                 int res;
20
21                 bio_init(&reply, rdata, sizeof(rdata), 4);
22                 bio_init_from_txn(&msg, txn);
23                 res = func(bs, txn, &msg, &reply);
24                 binder_send_reply(bs, &reply, txn->data, res);
25             }
26             ptr += sizeof(*txn) / sizeof(uint32_t);
27             break;
28         }
29         ......
30         }
31     }
32
33     return r;
34 }
```

在分析函数binder_parse的实现之前，我们首先介绍三个结构体：binder_object、binder_txn和binder_io，它们的定义如下所示。

frameworks/base/cmds/servicemanager/binder.h

```
01 struct binder_object
02 {
03     uint32_t type;
04     uint32_t flags;
05     void *pointer;
06     void *cookie;
07 };
08
09 struct binder_txn
10 {
11     void *target;
12     void *cookie;
13     uint32_t code;
14     uint32_t flags;
15
```

```
16     uint32_t sender_pid;
17     uint32_t sender_euid;
18
19     uint32_t data_size;
20     uint32_t offs_size;
21     void *data;
22     void *offs;
23 };
24
25 struct binder_io
26 {
27     char *data;            /* pointer to read/write from */
28     uint32_t *offs;        /* array of offsets */
29     uint32_t data_avail;   /* bytes available in data buffer */
30     uint32_t offs_avail;   /* entries available in offsets array */
31
32     char *data0;           /* start of data buffer */
33     uint32_t *offs0;       /* start of offsets buffer */
34     uint32_t flags;
35     uint32_t unused;
36 };
```

结构体binder_txn用来描述进程间通信数据，它等同于在前面5.1.1小节中介绍的结构体binder_transaction_data。结构体binder_object用来描述进程间通信数据中的一个Binder对象，它等同于在前面5.1.1小节中介绍的结构体flat_binder_object。结构体binder_io是用来解析进程间通信数据的，它的作用类似于Binder库中的Parcel类。

在结构体binder_io中，成员变量data和data0指向进程间通信数据中的数据缓冲区，成员变量offs和offs0指向进程间通信数据中的偏移数组，成员变量data_avail和offs_avail分别用来描述数据缓冲区data0和偏移数组offs0中还有多少内容是未被解析的。

在解析进程间通信数据的过程中，结构体binder_io的成员变量data0和offs0始终指向数据缓冲区和偏移数组的起始地址，而成员变量data和offs指向当前正在解析的数据缓冲区和偏移数组的地址。

结构体binder_io的成员变量unused保留给以后使用。成员变量flags用来描述数据缓冲区的属性，如下所示。

frameworks/base/cmds/servicemanager/binder.c

```
1 #define BIO_F_SHARED    0x01  /* needs to be buffer freed */
2 #define BIO_F_OVERFLOW  0x02  /* ran out of space */
3 #define BIO_F_IOERROR   0x04
4 #define BIO_F_MALLOCED  0x08  /* needs to be free()'d */
```

宏BIO_F_SHARED表示结构体binder_io内部的数据缓冲区是一块在内核空间分配的内核缓冲区，并且可以通过用户空间地址来共享访问。当进程使用完成这个数据缓冲区之后，它就必须使用BC_FREE_BUFFER命令协议来通知Binder驱动程序释放相应的内核缓冲区。

宏BIO_F_MALLOCED表示结构体binder_io内部的数据缓冲区是通过函数malloc来分配的，即它指向的是一块在用户空间分配的缓冲区。当进程使用完成这个数据缓冲区之后，直接调用函数free释放它即可。

宏BIO_F_OVERFLOW和BIO_F_IOERROR表示两个错误代码，其中，BIO_F_OVERFLOW表示数据溢出，即上次要求从结构体binder_io读出的数据的大小超出了其内部的数据缓冲区的大小；BIO_F_IOERROR表示上次从结构体binder_io读数据时发生了IO错误。

回到函数binder_parse中，我们只需关注它处理BR_TRANSACTION返回协议的过程，因为前面它从Binder驱动程序中接收到了一个BR_TRANSACTION返回协议。第8行从缓冲区ptr读出BR_TRANSACTION返回协议代码之后，接着第13行又将BR_TRANSACTION返回协议内容读取到一个binder_txn结构体

txn中。

第17行和第18行代码分别定义了两个binder_io结构体msg和reply，其中，msg用来解析从Binder驱动程序读取回来的进程间通信数据；而reply用来将进程间通信结果数据保存到缓冲区rdata中，以便后面可以将它返回给Binder驱动程序。它们分别使用函数bio_init和bio_init_from_txn来初始化，接下来我们就分析这两个函数的实现。

函数bio_init的实现如下所示。

frameworks/base/cmds/servicemanager/binder.c

```
01 void bio_init(struct binder_io *bio, void *data,
02                uint32_t maxdata, uint32_t maxoffs)
03 {
04     uint32_t n = maxoffs * sizeof(uint32_t);
05
06     if (n > maxdata) {
07         bio->flags = BIO_F_OVERFLOW;
08         bio->data_avail = 0;
09         bio->offs_avail = 0;
10         return;
11     }
12
13     bio->data = bio->data0 = data + n;
14     bio->offs = bio->offs0 = data;
15     bio->data_avail = maxdata - n;
16     bio->offs_avail = maxoffs;
17     bio->flags = 0;
18 }
```

第一个参数bio指向要初始化的binder_io结构体；第二个参数data指向binder_io结构体bio内部所使用的缓冲区；第三个参数maxdata用来描述缓冲区data的大小；第四个参数maxoffs描述binder_io结构体bio内部的偏移数组的大小。

第6行的if语句判断binder_io结构体bio所需要的偏移数组的大小是否大于缓冲区data的大小。如果是，那么就说明缓冲区data的大小不足，因此，第10行就直接返回了。

第13行和第14行代码将缓冲区data划分成两部分，其中一部分用于binder_io结构体bio的数据缓冲区，另一部分用于binder_io结构体bio的偏移数组。最后，第15行和第16行代码设置binder_io结构体bio的数据缓冲区和偏移数组的可用大小。

函数bio_init_from_txn的实现如下所示。

frameworks/base/cmds/servicemanager/binder.c

```
1 void bio_init_from_txn(struct binder_io *bio, struct binder_txn *txn)
2 {
3     bio->data = bio->data0 = txn->data;
4     bio->offs = bio->offs0 = txn->offs;
5     bio->data_avail = txn->data_size;
6     bio->offs_avail = txn->offs_size / 4;
7     bio->flags = BIO_F_SHARED;
8 }
```

第一个参数bio指向要初始化的binder_io结构体；第二个参数txn指向一个binder_txn 结构体txn，它里面包含了binder_io结构体bio要解析的数据缓冲区和偏移数组。

第3行和第4行代码用来设置binder_io结构体bio的数据缓冲区和偏移数组，它们分别指向binder_txn结构体txn中的数据缓冲区和偏移数组。接着第5行和第6行代码用来设置binder_io结构体bio的数据缓冲区和偏移数组的可用大小。

最后，第7行代码将binder_io结构体bio的成员变量flags的值设置为BIO_F_SHARED，表示它内部的数据缓冲区和偏移数组是在内核空间分配的。

回到函数binder_parse中，初始化完成binder_io结构体msg和reply之后，第23行就调用函数func来处理保存在binder_io结构体msg中的BR_TRANSACTION返回协议，并且将处理结果保存在binder_io结构体reply中。最后，第24行调用函数binder_send_reply将进程间通信结果，即binder_io结构体reply返回给Binder驱动程序。在接下来的5.8.1.4小节中，我们再详细分析函数binder_send_reply是如何将进程间通信结果返回给Binder驱动程序的。

从前面5.6.3小节中的内容可以知道，函数指针func指向的是Service Manager进程中的函数svcmgr_handler，它的实现如下所示。

frameworks/base/cmds/servicemanager/service_manager.c

```
01 int svcmgr_handler(struct binder_state *bs,
02                    struct binder_txn *txn,
03                    struct binder_io *msg,
04                    struct binder_io *reply)
05 {
06     struct svcinfo *si;
07     uint16_t *s;
08     unsigned len;
09     void *ptr;
10     uint32_t strict_policy;
11
12     ......
13
14     if (txn->target != svcmgr_handle)
15         return -1;
16
17     // Equivalent to Parcel::enforceInterface(), reading the RPC
18     // header with the strict mode policy mask and the interface name.
19     // Note that we ignore the strict_policy and don't propagate it
20     // further (since we do no outbound RPCs anyway).
21     strict_policy = bio_get_uint32(msg);
22     s = bio_get_string16(msg, &len);
23     if ((len != (sizeof(svcmgr_id) / 2)) ||
24         memcmp(svcmgr_id, s, sizeof(svcmgr_id))) {
25         fprintf(stderr,"invalid id %s\n", str8(s));
26         return -1;
27     }
28
29     switch(txn->code) {
30     ......
31     case SVC_MGR_ADD_SERVICE:
32         s = bio_get_string16(msg, &len);
33         ptr = bio_get_ref(msg);
34         if (do_add_service(bs, s, len, ptr, txn->sender_euid))
35             return -1;
36         break;
37     ......
38     }
39
40     bio_put_uint32(reply, 0);
41     return 0;
42 }
```

第14行检查从Binder驱动程序传进来的目标Binder本地对象txn->target是否指向在Service Manager中定义的虚拟Binder本地对象svcmgr_handle。如果不是，就说明Service Manager正在处理一个非法的进程间通信请求，因此，第15行就直接返回一个错误码-1给调用者。

第21行到第27行代码检查Binder进程间通信请求头是否合法。一个合法的Binder进程间通信请求头由一个Strict Mode Policy和一个服务接口描述符组成。Service Manager的服务接口描述符定义为svcmgr_id，它的定义如下所示。

frameworks/base/cmds/servicemanager/service_manager.c

```
1 uint16_t svcmgr_id[] = {
2     'a','n','d','r','o','i','d','.','o','s','.',
3     'I','S','e','r','v','i','c','e','M','a','n','a','g','e','r'
4 };
```

它是一个uint16_t类型的字符数组，因此在计算svcmgr_id的字符串长度时，需要将数组的长度除以2。

回到函数svcmgr_handler中，从第17行到第20行的注释可以知道，Service Manager忽略了Binder进程间通信请求头的Strict Mode Policy值，但是它需要验证传递过来的服务接口描述符是否等于svcmgr_id。如果不相等，就说明这是一个非法的进程间通信请求，因此，第26行就直接返回一个错误码-1给调用者。

由于FregServer进程请求Service Manager执行的操作是注册Service组件FregService，对应的操作代码为SVC_MGR_ADD_SERVICE，因此，接下来我们就只分析binder_txn 结构体txn的成员变量code的值等于SVC_MGR_ADD_SERVICE的情况。

第32行从binder_io结构体msg的数据缓冲区中取出要注册的Service组件的名称，接着第33行调用函数bio_get_ref从binder_io结构体msg的数据缓冲区中获得一个Binder引用对象的句柄值，这个Binder引用对象是在Binder驱动程序中创建的，它引用了即将要注册的Serivce组件。

函数bio_get_ref的实现如下所示。

frameworks/base/cmds/servicemanager/binder.c

```
01 void *bio_get_ref(struct binder_io *bio)
02 {
03     struct binder_object *obj;
04
05     obj = _bio_get_obj(bio);
06     if (!obj)
07         return 0;
08
09     if (obj->type == BINDER_TYPE_HANDLE)
10         return obj->pointer;
11
12     return 0;
13 }
```

函数bio_get_ref通过调用函数_bio_get_obj从binder_io结构体bio中取出一个binder_object结构体obj。如果取出来的binder_object结构体obj的类型为BINDER_TYPE_HANDLE，那么就将它的成员变量pointer返回给调用者。类型为BINDER_TYPE_HANDLE的binder_object结构体的成员变量pointer保存的是一个由Binder驱动程序创建的Binder引用对象的句柄值，这个Binder引用对象引用了即将要注册的Service组件。

函数_bio_get_obj的实现如下所示。

frameworks/base/cmds/servicemanager/binder.c

```
01 static struct binder_object *_bio_get_obj(struct binder_io *bio)
02 {
03     unsigned n;
04     unsigned off = bio->data - bio->data0;
05
06         /* TODO: be smarter about this? */
07     for (n = 0; n < bio->offs_avail; n++) {
08         if (bio->offs[n] == off)
09             return bio_get(bio, sizeof(struct binder_object));
10     }
11
```

```
12      bio->data_avail = 0;
13      bio->flags |= BIO_F_OVERFLOW;
14      return 0;
15  }
```

函数_bio_get_obj从binder_io结构体bio的数据缓冲区的当前位置读取一个binder_object结构体。第7行到第10行的for循环检查binder_io结构体bio的数据缓冲区的当前位置保存的是否是一个binder_object结构体，方法是检查binder_io结构体bio的偏移数组中是否有一个元素的值刚好等于binder_io结构体bio的数据缓冲区的当前位置。合法性检查通过之后，第9行就继续调用函数bio_get将该binder_object结构体读取出来。

函数bio_get的实现如下所示。

frameworks/base/cmds/servicemanager/binder.c

```
01 static void *bio_get(struct binder_io *bio, uint32_t size)
02 {
03      size = (size + 3) & (~3);
04
05      if (bio->data_avail < size){
06          bio->data_avail = 0;
07          bio->flags |= BIO_F_OVERFLOW;
08          return 0;
09      }  else {
10          void *ptr = bio->data;
11          bio->data += size;
12          bio->data_avail -= size;
13          return ptr;
14      }
15 }
```

参数size表示要从binder_io结构体bio的数据缓冲区的当前位置读取的数据对象的大小，第3行将它对齐到4个字节边界。从前面的调用过程可以知道，传进来的参数size的大小等于结构体binder_object的大小，它本身就是对齐到4个字节边界的，因此，第3行其实并没有改变参数size的大小。

第5行的if语句检查binder_io结构体bio的数据缓冲区的剩余未解析字节数data_avail是否小于要求读取的字节数size。如果是，就说明出错了，因此，第7行就将binder_io结构体bio的成员变量flags的BIO_F_OVERFLOW位设置为1，接着第8行返回一个地址值0给调用者。

如果binder_io结构体bio的数据缓冲区的剩余未解析字节数data_avail大于或者等于要求读取的字节数size，即第5行的if语句为false，那么第10行代码就将binder_io结构体bio的数据缓冲区data的当前位置保存在变量ptr中。接着第11行代码再将它往前推进size个字节，并且第12行代码将binder_io结构体bio的数据缓冲区的剩余未读取字节数减少size个字节，最后第13行将变量ptr的值返回给调用者。

回到函数svcmgr_handler中，获得了用来描述即将要注册的Service组件的一个句柄值ptr之后，接着第34行就调用函数do_add_service将这个Service组件注册到Service Manager中。

函数do_add_service的实现如下所示。

frameworks/base/cmds/servicemanager/service_manager.c

```
01 int do_add_service(struct binder_state *bs,
02                     uint16_t *s, unsigned len,
03                     void *ptr, unsigned uid)
04 {
05      struct svcinfo *si;
06      ......
07
08      if (!svc_can_register(uid, s)) {
09          ......
10          return -1;
11      }
```

```
12
13      si = find_svc(s, len);
14      if (si) {
15          if (si->ptr) {
16              ......
17              return -1;
18          }
19          si->ptr = ptr;
20      } else {
21          si = malloc(sizeof(*si) + (len + 1) * sizeof(uint16_t));
22          ......
23
24          si->ptr = ptr;
25          si->len = len;
26          memcpy(si->name, s, (len + 1) * sizeof(uint16_t));
27          si->name[len] = '\0';
28          si->death.func = svcinfo_death;
29          si->death.ptr = si;
30          si->next = svclist;
31          svclist = si;
32      }
33
34      binder_acquire(bs, ptr);
35      binder_link_to_death(bs, ptr, &si->death);
36      return 0;
37  }
```

参数s表示要注册的Service组件的名称；参数uid表示请求Service Manager注册Service组件的进程的用户ID。第8行调用函数svc_can_register来检查用户ID为uid的进程是否有权限请求Service Manager注册一个名称为s的Serivce组件。如果没有，第10行就直接返回错误码-1给调用者。

函数svc_can_register的实现如下所示。

frameworks/base/cmds/servicemanager/service_manager.c

```
01 int svc_can_register(unsigned uid, uint16_t *name)
02 {
03     unsigned n;
04
05     if ((uid == 0) || (uid == AID_SYSTEM))
06         return 1;
07
08     for (n = 0; n < sizeof(allowed) / sizeof(allowed[0]); n++)
09         if ((uid == allowed[n].uid) && str16eq(name, allowed[n].name))
10             return 1;
11
12     return 0;
13 }
```

将Service组件注册到Service Manager是一种特权，即不是所有的进程都可以将Service组件注册Service Manager中。Service Manager定义了一个全局数组allowed，它定义了哪些进程可以注册什么名称的Service组件，它的定义如下所示。

frameworks/base/cmds/servicemanager/service_manager.c

```
01 /* TODO:
02  * These should come from a config file or perhaps be
03  * based on some namespace rules of some sort (media
04  * uid can register media.*, etc)
05  */
06 static struct {
07     unsigned uid;
08     const char *name;
09 } allowed[] = {
```

```
10 #ifdef LVMX
11     { AID_MEDIA, "com.lifevibes.mx.ipc" },
12 #endif
13     { AID_MEDIA, "media.audio_flinger" },
14     { AID_MEDIA, "media.player" },
15     { AID_MEDIA, "media.camera" },
16     { AID_MEDIA, "media.audio_policy" },
17     { AID_DRMIO, "drm.drmIOService" },
18     { AID_DRM,   "drm.drmManager" },
19     { AID_NFC,   "nfc" },
20     { AID_RADIO, "radio.phone" },
21     { AID_RADIO, "radio.sms" },
22     { AID_RADIO, "radio.phonesubinfo" },
23     { AID_RADIO, "radio.simphonebook" },
24 /* TODO: remove after phone services are updated: */
25     { AID_RADIO, "phone" },
26     { AID_RADIO, "sip" },
27     { AID_RADIO, "isms" },
28     { AID_RADIO, "iphonesubinfo" },
29     { AID_RADIO, "simphonebook" },
30 };
```

数组allowed中的每一个元素都使用一个匿名结构体来描述。在这个匿名结构体中，成员变量uid用来描述一个用户ID；成员变量name用来描述一个服务名称，即只有用户ID为uid的进程才可以注册名称为name的Service组件。例如，匿名结构体{AID_MEDIA, "com.lifevibes.mx.ipc"}表示只有用户ID为AID_MEDIA的进程才可以注册名称为“com.lifevibes.mx.ipc”的Service组件。

回到函数svc_can_register中，第8行到第10行的for循环检查参数uid和name是否对应于数组allowed中的某一个元素。如果是，就说明用户ID为uid的进程有权限将名称为name的Service组件注册到Service Manager中，因此，第10行就返回1给调用者，表示可以注册一个名称为name的Service组件。

有一种特殊情况，即请求注册Service组件的进程的用户ID等于0或者AID_SYSTEM的进程不受数组allowed的限制，它可以随意将Service组件注册到Service Manager中，因为它们是系统进程，具有比一般用户进程更高的权限。因此，在函数svc_can_register中，第5行的if语句如果发现参数uid的值等于0或者AID_SYSTEM，第10行就马上返回1给调用者，表示可以注册一个名称为name的Service组件。

回到函数do_add_service中，通过了注册Service组件的权限检查之后，接着第13行继续调用函数find_svc来检查服务名称s是否被已经注册了的Service组件使用了。

函数find_svc的实现如下所示。

frameworks/base/cmds/servicemanager/service_manager.c

```
01 struct svcinfo *find_svc(uint16_t *s16, unsigned len)
02 {
03     struct svcinfo *si;
04
05     for (si = svclist; si; si = si->next) {
06         if ((len == si->len) &&
07             !memcmp(s16, si->name, len * sizeof(uint16_t))) {
08             return si;
09         }
10     }
11     return 0;
12 }
```

在Service Manager中，每一个被注册了的Service组件都使用一个svcinfo结构体来描述，并且保存在一个全局队列svclist中。

结构体svcinfo和全局队列svclist的定义如下所示。

frameworks/base/cmds/servicemanager/service_manager.c

```
01 struct svcinfo
02 {
03     struct svcinfo *next;
04     void *ptr;
05     struct binder_death death;
06     unsigned len;
07     uint16_t name[0];
08 };
09
10 struct svcinfo *svclist = 0;
```

在结构体svcinfo中，成员变量next用来指向下一个svcinfo结构体；成员变量ptr是一个句柄值，用来描述一个注册了的Service组件；成员变量name和len分别用来描述已经注册了的Service组件的名称及其长度；成员变量death指向一个binder_death结构体，用来描述一个死亡接收通知。

结构体binder_death的定义如下所示。

frameworks/base/cmds/servicemanager/service_manager.c

```
1 struct binder_death {
2     void (*func)(struct binder_state *bs, void *ptr);
3     void *ptr;
4 };
```

在结构体binder_death中，成员变量ptr指向其宿主svcinfo结构体；成员变量func是一个函数指针，当与其宿主svcinfo结构体所对应的Service组件死亡时，该函数指针所指向的函数就会被调用，以便可以获得相应的Service组件的死亡通知。

回到函数find_svc中，第5行到第10行的for循环依次检查全局队列svclist中的已注册Service组件列表。如果发现参数s16所描述的Service组件名称已经被使用了，那么第8行就会将与参数s16所对应的一个svcinfo结构体返回给调用者。

回到函数do_add_service中，如果指定要注册的Service组件的名称s已经被使用了，即第14行的if语句为true，那么第15行就继续检查与名称s关联的svcinfo结构体si的成员变量ptr的值是否等于NULL。如果不等于，即第15行的if语句为true，那么就说明该名称s已经被使用了，因此，第17行直接返回一个错误码-1给调用者；否则，第19行就将svcinfo结构体si的成员变量ptr的值修改为参数ptr的值，即将它修改为一个引用了要注册的Service组件的Binder引用对象的句柄值。

如果指定要注册的Service组件的名称s没有被使用，即第14行的if语句为false，那么第21行到第31行代码就会创建一个svcinfo结构体来描述要注册的Service组件，并且将它添加到全局队列svclist中。

至此，一个进程就成功地将一个Service组件注册到Service Manager中了。从Service Manager的角度来看，由于它引用了新注册的Service组件，因此，第34行需要调用函数binder_acquire来增加相应的Binder引用对象的引用计数值，避免它过早地被销毁。函数binder_acquire实际上是使用BC_ACQUIRE命令协议来通知Binder驱动程序增加相应的Binder引用对象的引用计数的。BC_ACQUIRE命令协议的处理过程，可以参考前面5.4小节所介绍的Binder对象引用计数技术。由于新注册的Service组件可能会意外地死亡，因此，第35行需要调用函数binder_link_to_death向Binder驱动程序注册一个Binder本地对象死亡接收通知，以便Service Manager可以在该Service组件死亡时采取相应的处理措施。Binder本地对象死亡接收通知的注册过程，可以参考前面5.5小节所介绍的Binder对象死亡通知机制。

函数do_add_service成功地将一个Service组件注册到Service Manager中之后，就返回到函数svcmgr_handler中，接着第40行就调用函数bio_put_uint32将注册成功代码0写入到binder_io结构体reply中，以便后面可以将它返回给请求注册Service组件的进程。

5.8.1.4 发送和处理BC_REPLY命令协议

从前面5.8.1.3小节的内容可以知道，Service Manager成功地将一个Service组件注册到内部的Service组件列表svclist中之后，接着就会调用函数binder_send_reply将Service组件注册结果返回给Binder驱动程序，Binder驱动程序再将该结果返回给请求注册Service组件的进程。

函数binder_send_reply的实现如下所示。

frameworks/base/cmds/servicemanager/binder.c

```
01 void binder_send_reply(struct binder_state *bs,
02                         struct binder_io *reply,
03                         void *buffer_to_free,
04                         int status)
05 {
06     struct {
07         uint32_t cmd_free;
08         void *buffer;
09         uint32_t cmd_reply;
10         struct binder_txn txn;
11     } __attribute__((packed)) data;
12
13     data.cmd_free = BC_FREE_BUFFER;
14     data.buffer = buffer_to_free;
15     data.cmd_reply = BC_REPLY;
16     data.txn.target = 0;
17     data.txn.cookie = 0;
18     data.txn.code = 0;
19     if (status) {
20         data.txn.flags = TF_STATUS_CODE;
21         data.txn.data_size = sizeof(int);
22         data.txn.offs_size = 0;
23         data.txn.data = &status;
24         data.txn.offs = 0;
25     } else {
26         data.txn.flags = 0;
27         data.txn.data_size = reply->data - reply->data0;
28         data.txn.offs_size = ((char*) reply->offs) - ((char*) reply->offs0);
29         data.txn.data = reply->data0;
30         data.txn.offs = reply->offs0;
31     }
32     binder_write(bs, &data, sizeof(data));
33 }
```

参数reply指向一个binder_io结构体，它内部包含了进程间通信结果数据；参数buffer_to_free是一个用户空间地址，它指向了一块用来传输进程间通信数据的内核缓冲区；参数status用来描述Service Manager是否成功地处理了一个进程间通信请求，即是否成功地注册了一个Service组件。

第6行到第11行代码定义了一个匿名结构体data，用来描述一个BC_FREE_BUFFER和一个BC_REPLY命令协议，分别用成员变量cmd_free和cmd_reply来表示。在前面的5.1.1小节中介绍Binder命令协议BinderDriverCommandProtocol时提到，命令协议BC_FREE_BUFFER后面跟的通信数据是一个内核缓冲区的用户空间地址，它就保存在成员变量buffer中；而命令协议BC_REPLY后面跟的通信数据是一个binder_transaction_data结构体，即一个binder_txn结构体，它就保存在成员变量txn中。

第13行和第14行代码用来设置匿名结构体data中的BC_FREE_BUFFER命令协议内容；第15行到第31行代码用来设置匿名结构体data中的BC_REPLY命令协议内容。

如果参数status的值不等于0，即第19行的if语句为true，就说明Service Manager在处理一个进程间通信请求时，发生了错误，错误代码就为status。在这种情况下，Service Manager需要将该错误代码通过BC_REPLY命令协议返回给Binder驱动程序，以便Binder驱动程序可以继续将它返回给发出该进程

间通信请求的Client进程，如第20行到第24行代码所示。

如果参数status的值等于0，即第19行的if语句为false，就说明Service Manager成功地处理了一个进程间通信请求。在这种情况下，进程间通信结果就保存在binder_io结构体reply中，因此，第26行到第30行代码就将binder_io结构体reply的数据缓冲区和偏移数组设置到匿名结构体data中，以便可以通过BC_REPLY命令协议返回给Binder驱动程序，Binder驱动程序再将它们返回给发出该进程间通信请求的Client进程。

最后，第32行调用函数binder_write将匿名结构体data中的BC_FREE_BUFFER和BC_REPLY命令协议发送给Binder驱动程序。

函数binder_write的实现如下所示。

frameworks/base/cmds/servicemanager/binder.c

```
01 int binder_write(struct binder_state *bs, void *data, unsigned len)
02 {
03     struct binder_write_read bwr;
04     int res;
05     bwr.write_size = len;
06     bwr.write_consumed = 0;
07     bwr.write_buffer = (unsigned) data;
08     bwr.read_size = 0;
09     bwr.read_consumed = 0;
10     bwr.read_buffer = 0;
11     res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
12     if (res < 0) {
13         fprintf(stderr,"binder_write: ioctl failed (%s)\n",
14                 strerror(errno));
15     }
16     return res;
17 }
```

函数binder_write是通过IO控制命令BINDER_WRITE_READ来将BC_FREE_BUFFER和BC_REPLY命令协议发送给Binder驱动程序的，因此，第3行定义了一个binder_write_read结构体bwr。第7行将binder_write_read结构体bwr的输入缓冲区write_buffer设置为由参数data所描述的一块用户空间缓冲区。从前面的调用过程可以知道，参数data所描述的一块用户空间缓冲区包含了一个BC_FREE_BUFFER和一个BC_REPLY命令协议。

注意

binder_write_read结构体bwr的输入缓冲区read_buffer及其长度都被设置为0，因此，当Service Manager通过IO控制命令BINDER_WRITE_READ进入到Binder驱动程序时，Binder驱动程序调用函数binder_thread_write处理完成binder_write_read结构体bwr的输出缓冲区中的BC_FREE_BUFFER和BC_REPLY命令协议之后，就会马上返回到Service Manager的用户空间。

接下来，我们就分别分析Binder驱动程序处理BC_FREE_BUFFER和BC_REPLY命令协议的过程。

BC_FREE_BUFFER命令协议是在函数binder_thread_write中处理的，如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 int
02 binder_thread_write(struct binder_proc *proc, struct binder_thread *thread,
03                      void __user *buffer, int size, signed long *consumed)
04 {
05         uint32_t cmd;
06         void __user *ptr = buffer + *consumed;
07         void __user *end = buffer + size;
08
09         while (ptr < end && thread->return_error == BR_OK) {
```

```
10                  if (get_user(cmd, (uint32_t __user *)ptr))
11                          return -EFAULT;
12                  ptr += sizeof(uint32_t);
13                  ......
14                  switch (cmd) {
15                  ......
16                  case BC_FREE_BUFFER: {
17                          void __user *data_ptr;
18                          struct binder_buffer *buffer;
19
20                          if (get_user(data_ptr, (void * __user *)ptr))
21                                  return -EFAULT;
22                          ptr += sizeof(void *);
23
24                          buffer = binder_buffer_lookup(proc, data_ptr);
25                          if (buffer == NULL) {
26                                  ......
27                                  break;
28                          }
29                          if (!buffer->allow_user_free) {
30                                  ......
31                                  break;
32                          }
33                          ......
34
35                          if (buffer->transaction) {
36                                  buffer->transaction->buffer = NULL;
37                                  buffer->transaction = NULL;
38                          }
39                          if (buffer->async_transaction && buffer->target_node) {
40                                  BUG_ON(!buffer->target_node->has_async_transaction);
41                                  if (list_empty(&buffer->target_node->async_todo))
42                                      buffer->target_node->has_async_transaction = 0;
43                                  else
44                                      list_move_tail(buffer->target_node->async_todo.next, &thread->todo);
45                          }
46                          binder_transaction_buffer_release(proc, buffer, NULL);
47                          binder_free_buf(proc, buffer);
48                          break;
49                  }
50                  ......
51                  }
52                  *consumed = ptr - buffer;
53          }
54          return 0;
55  }
```

第20行得到要释放的内核缓冲区的用户空间地址，保存在变量data_ptr中，接着第24行调用函数binder_buffer_lookup在进程proc中找到与用户空间地址data_ptr对应的内核缓冲区buffer。如果找到的内核缓冲区buffer为NULL，或者它的成员变量allow_user_free的值等于0，即不允许进程从用户空间发送BC_FREE_BUFFER命令协议来释放，那么函数就不往下处理了。

第35行的if语句检查内核缓冲区buffer的成员变量transaction是否不等于NULL。如果是，就说明该内核缓冲区是分配给一个binder_transaction结构体使用的。现在既然要释放这个内核缓冲区，第36行就将该binder_transaction结构体的成员变量buffer清空，并且第37行将该内核缓冲区的成员变量transaction也清空。

第39行的if语句检查内核缓冲区buffer是否是分配给一个Binder实体对象用来处理异步事务的。如果是，就需要对它的目标Binder实体对象的异步事务队列进行处理。因为当一个用于异步事务处理的内核缓冲区被用户空间通过BC_FREE_BUFFER命令协议释放时，就说明该异步事务已经处理完成

了，因此，就需要着手处理下一个异步事务了。第41行检查内核缓冲区buffer的目标Binder实体对象target_node的异步事务队列async_todo是否为空。如果是，就说明该Binder实体对象没有正在等待处理的异步事务，因此，第42行只需要将它的成员变量has_async_transaction的值设置为0即可。如果内核缓冲区buffer的目标Binder实体对象target_node的异步事务队列async_todo不为空，即第41行的if语句为false，那么第44行就将该异步事务队列async_todo中的下一个异步事务移至目标线程thread的todo队列中去等待处理。这样就可以保证一个Binder实体对象的所有异步事务都是串行处理的，相当于给予Binder实体对象更高的优先级来处理同步事务。

在前面的5.8.1.2小节中分析函数binder_transaction的实现时提到，分配给一个binder_transaction结构体使用的内核缓冲区是用来保存进程间通信数据的。这些进程间通信数据中可能包含有Binder对象，Binder驱动程序将这些进程间通信数据传输给目标进程时，曾经增加过它们所对应的Binder实体对象或者Binder引用对象的引用计数，避免这些Binder实体对象或者Binder引用对象在该内核缓冲区使用期间被销毁。现在既然内核缓冲区buffer已经使用完成了，第46行就调用函数binder_transaction_buffer_release来减少它里面的Binder实体对象或者Binder引用对象的引用计数。此外，如果该内核缓冲区buffer的成员变量target_node的值不等于NULL，即它是分配给一个Binder实体对象使用的，那么函数binder_transaction_buffer_release也会减少该Binder实体对象的引用计数，因为之前曾经增加过它的引用计数，避免它在内核缓冲区buffer的使用期间被销毁。

最后，第47行调用函数binder_free_buf来释放内核缓冲区buffer。在前面的5.1.5.2小节中分析Binder驱动程序中的内核缓冲区释放过程时，我们已经分析过函数binder_free_buf的实现了，因此，接下来我们主要分析函数binder_transaction_buffer_release的实现。

函数binder_transaction_buffer_release的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static void
02 binder_transaction_buffer_release(struct binder_proc *proc, struct binder_buffer *buffer, size_t *failed_at)
03 {
04         size_t *offp, *off_end;
05         int debug_id = buffer->debug_id;
06         ......
07
08         if (buffer->target_node)
09                 binder_dec_node(buffer->target_node, 1, 0);
10
11         offp = (size_t *)(buffer->data + ALIGN(buffer->data_size, sizeof(void *)));
12         if (failed_at)
13                 off_end = failed_at;
14         else
15                 off_end = (void *)offp + buffer->offsets_size;
16         for (; offp < off_end; offp++) {
17                 struct flat_binder_object *fp;
18                 ......
19                 fp = (struct flat_binder_object *)(buffer->data + *offp);
20                 switch (fp->type) {
21                 case BINDER_TYPE_BINDER:
22                 case BINDER_TYPE_WEAK_BINDER: {
23                         struct binder_node *node = binder_get_node(proc, fp->binder);
24                         ......
25                         binder_dec_node(node, fp->type == BINDER_TYPE_BINDER, 0);
26                 } break;
27                 case BINDER_TYPE_HANDLE:
28                 case BINDER_TYPE_WEAK_HANDLE: {
29                         struct binder_ref *ref = binder_get_ref(proc, fp->handle);
30                         ......
31                         binder_dec_ref(ref, fp->type == BINDER_TYPE_HANDLE);
```

```
32                  } break;
33                  ......
34                  }
35          }
36 }
```

第8行检查即将要释放的内核缓冲区buffer是否是分配给一个Binder实体对象使用的，即它的成员变量target_node的值是否不等于NULL。如果是，那么第9行就调用函数binder_dec_node来减少它的引用计数。

第16行到第35行的for循环依次遍历即将要释放的内核缓冲区buffer中的Binder对象，并且减少它们所对应的Binder实体对象或者Binder引用对象的引用计数。

至此，BC_FREE_BUFFER命令协议的处理过程就分析完成了。接下来，我们继续分析BC_REPLY命令协议的处理过程。

BC_REPLY命令协议是在函数binder_transaction中处理的，如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static void
02 binder_transaction(struct binder_proc *proc, struct binder_thread *thread,
03         struct binder_transaction_data *tr, int reply)
04 {
05         struct binder_transaction *t;
06         struct binder_work *tcomplete;
07         ......
08         struct binder_thread *target_thread = NULL;
09         struct binder_node *target_node = NULL;
10         struct list_head *target_list;
11         wait_queue_head_t *target_wait;
12         struct binder_transaction *in_reply_to = NULL;
13         ......
14         uint32_t return_error;
15
16         if (reply) {
17                 in_reply_to = thread->transaction_stack;
18                 if (in_reply_to == NULL) {
19                         ......
20                         return_error = BR_FAILED_REPLY;
21                         goto err_empty_call_stack;
22                 }
23
24                 binder_set_nice(in_reply_to->saved_priority);
25                 if (in_reply_to->to_thread != thread) {
26                         ......
27                         return_error = BR_FAILED_REPLY;
28                         in_reply_to = NULL;
29                         goto err_bad_call_stack;
30                 }
31                 thread->transaction_stack = in_reply_to->to_parent;
32                 target_thread = in_reply_to->from;
33                 if (target_thread == NULL) {
34                         return_error = BR_DEAD_REPLY;
35                         goto err_dead_binder;
36                 }
37                 if (target_thread->transaction_stack != in_reply_to) {
38                         ......
39                         return_error = BR_FAILED_REPLY;
40                         in_reply_to = NULL;
41                         target_thread = NULL;
42                         goto err_dead_binder;
43                 }
44                 ......
```

```
45          } else {
46                  ......
47          }
48          if (target_thread) {
49                  ......
50                  target_list = &target_thread->todo;
51                  target_wait = &target_thread->wait;
52          } else {
53                  ......
54          }
55          ......
56
57          t = kzalloc(sizeof(*t), GFP_KERNEL);
58          ......
59
60          tcomplete = kzalloc(sizeof(*tcomplete), GFP_KERNEL);
61          ......
62
63          if (reply) {
64                  BUG_ON(t->buffer->async_transaction != 0);
65                  binder_pop_transaction(target_thread, in_reply_to);
66          } else if (!(t->flags & TF_ONE_WAY)) {
67                  ......
68          } else {
69                  ......
70          }
71          t->work.type = BINDER_WORK_TRANSACTION;
72          list_add_tail(&t->work.entry, target_list);
73          tcomplete->type = BINDER_WORK_TRANSACTION_COMPLETE;
74          list_add_tail(&tcomplete->entry, &thread->todo);
75          if (target_wait)
76                  wake_up_interruptible(target_wait);
77          return;
78
79          ......
80  }
```

函数binder_transaction在处理BC_REPLY命令协议时，传进来的参数reply的值就等于1，以便可以区别于BC_TRANSACTION命令协议。

第17行到第44行代码用来找到之前请求与线程thread进行进程间通信的线程。找到了这个目标线程target_thread之后，Binder驱动程序就可以向它发送一个BR_REPLY返回协议，以便将进程间通信结果返回给它。

从前面5.8.1.3小节的内容可以知道，当Binder驱动程序分发一个进程间通信请求给一个线程处理时，就会将一个binder_transaction 结构体压入到它的事务堆栈中，因此，第17行就从线程thread的事务堆栈中将该binder_transaction 结构体取出来，并且保存在变量in_reply_to中。binder_transaction 结构体in_reply_to的成员变量from指向了之前请求与线程thread进行进程间通信的线程，因此，第32行就可以通过它来获得目标线程target_thread。

从前面5.8.1.3小节的内容还可以知道，当Binder驱动程序分发一个进程间通信请求给一个线程处理时，会相应地修改它的线程优先级，以便它可以按照需求来处理一个进程间通信请求。现在既然这个线程已经处理完成这个进程间通信请求了，第24行就调用函数binder_set_nice来恢复它原来的线程优先级。

在前面的5.1.1小节中介绍结构体binder_transaction时提到，它的成员变量to_parent指向了用来处理它的线程thread的下一个需要处理的事务，因此，第31行就将这个事务放在线程thread的事务堆栈transaction_stack的顶端，表示线程thread接下来要处理它。

找到了目标线程target_thread之后，第50行和第51行代码就分别将它的todo队列和wait等待队列作

为目标todo队列target_list和目标wait等待队列target_wait。

第57行和第60行代码分别创建了一个binder_transaction结构体t和一个binder_work结构体tcomplete，后面会分别将它们封装成一个BINDER_WORK_TRANSACTION和一个BINDER_WORK_TRANSACTION_COMPLETE类型的工作项添加到目标todo队列和线程thread的todo队列中。

第65行调用函数binder_pop_transaction从目标线程target_thread的事务堆栈中删除binder_transaction结构体in_reply_to，因为它所描述的一个事务已经处理完成了，因此，目标线程target_thread就不需要再保存它了。

函数binder_pop_transaction的实现如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static void
02 binder_pop_transaction(
03         struct binder_thread *target_thread, struct binder_transaction *t)
04 {
05         if (target_thread) {
06                 BUG_ON(target_thread->transaction_stack != t);
07                 BUG_ON(target_thread->transaction_stack->from != target_thread);
08                 target_thread->transaction_stack =
09                         target_thread->transaction_stack->from_parent;
10                 t->from = NULL;
11         }
12         t->need_reply = 0;
13         if (t->buffer)
14                 t->buffer->transaction = NULL;
15         kfree(t);
16         ......
17 }
```

每一个线程正在处理的事务都应该位于它的事务堆栈的顶端，而只有当该事务处理完成之后，才可以将它删除。因此，第6行就验证要删除的binder_transaction结构体t是否位于目标线程target_thread的事务堆栈transaction_stack的顶端。接着第7行又继续验证要删除的binder_transaction结构体t是否是由目标线程target_thread创建的，即binder_transaction结构体t所描述的事务是否是由目标线程target_thread发起的。这两个合法性检查都通过之后，第8行和第9行就可以将目标线程target_thread的事务堆栈transaction_stack中的下一个事务移到它的顶端，相当于将binder_transaction结构体t所描述的事务从目标线程target_thread的事务堆栈transaction_stack的顶端移除。最后，第15行将binder_transaction结构体t所占用的内存释放。

回到函数binder_transaction中，第71行将前面创建的binder_transaction结构体t的类型设置为BINDER_WORK_TRANSACTION。接着第72行将它封装成一个工作项添加到目标todo队列，即目标线程target_thread的todo队列中。最后第76行会通过目标wait等待队列将该线程唤醒，以便它可以处理刚才添加到todo队列的工作项。

第73行将前面创建的binder_work结构体tcomplete的类型设置为BINDER_WORK_TRANSACTION_COMPLETE。接着第74行将它封装成一个工作项添加到线程thread的todo队列中，当线程thread从Binder驱动程序返回到用户空间之前，就会对该工作项进行处理。

在前面的5.8.1.2小节中，我们已经分析过类型为BINDER_WORK_TRANSACTION_COMPLETE的工作项的处理过程，Binder驱动程序是通过向目标线程发送一个BR_TRANSACTION_COMPLETE返回协议来对它进行处理的。在上述情景中，目标线程即为Service Manager进程中的主线程，它处理完成Binder驱动程序给它发送的BR_TRANSACTION_COMPLETE返回协议之后，就彻底地处理完成一次进程间通信请求了。接下来它又会通过IO控制命令BINDER_WRITE_READ重新进入到Binder驱动程序中去等待新的进程间通信请求。

在这个情景中，目标线程target_thread即为FregServer进程中请求注册Service组件FregService的线程，在前面的5.8.1.2小节中，我们假设它正在Binder驱动程序的函数binder_thread_read中等待Service Manager完成对Service组件FregService的注册。现在既然Service Manager已经完成了对Service组件FregService的注册，并且将目标线程target_thread唤醒了，因此，接下来目标线程target_thread就会继续执行函数binder_thread_read来处理它的todo队列中的工作项。

5.8.1.5　发送和处理BR_REPLY返回协议

在前面的5.8.1.2小节中，我们假设请求注册Service组件FregService的线程thread正在Binder驱动程序的函数binder_thread_read中等待Service Manager完成注册操作。现在既然Service Manager已经完成了对Service组件FregService的注册，并且线程thread已经被Binder驱动程序唤醒，接下来它就会继续执行函数binder_thread_read来处理它的todo队列中的工作项了。

从前面5.8.1.4小节的内容可以知道，现在请求注册Service组件FregService的线程thread的todo队列中有一个类型为BINDER_WORK_TRANSACTION的工作项，因此，接下来我们就分析函数binder_thread_read处理这个工作项的过程。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static int
02 binder_thread_read(struct binder_proc *proc, struct binder_thread *thread,
03         void  __user *buffer, int size, signed long *consumed, int non_block)
04 {
05         void __user *ptr = buffer + *consumed;
06         void __user *end = buffer + size;
07         ......
08
09         while (1) {
10                 uint32_t cmd;
11                 struct binder_transaction_data tr;
12                 struct binder_work *w;
13                 struct binder_transaction *t = NULL;
14
15                 if (!list_empty(&thread->todo))
16                         w = list_first_entry(&thread->todo, struct binder_work, entry);
17                 else if (!list_empty(&proc->todo) && wait_for_proc_work)
18                         w = list_first_entry(&proc->todo, struct binder_work, entry);
19                 else {
20                         ......
21                 }
22
23                 ......
24
25                 switch (w->type) {
26                 case BINDER_WORK_TRANSACTION: {
27                         t = container_of(w, struct binder_transaction, work);
28                 } break;
29                 ......
30                 }
31                 ......
32
33                 if (t->buffer->target_node) {
34                         ......
35                 } else {
36                         tr.target.ptr = NULL;
37                         tr.cookie = NULL;
38                         cmd = BR_REPLY;
39                 }
```

```
40                  ......
41
42                  tr.data_size = t->buffer->data_size;
43                  tr.offsets_size = t->buffer->offsets_size;
44                  tr.data.ptr.buffer = (void *)t->buffer->data + proc->user_buffer_offset;
45                  tr.data.ptr.offsets = tr.data.ptr.buffer + ALIGN(t->buffer->data_size, sizeof(void *));
46
47                  if (put_user(cmd, (uint32_t __user *)ptr))
48                          return -EFAULT;
49                  ptr += sizeof(uint32_t);
50                  if (copy_to_user(ptr, &tr, sizeof(tr)))
51                          return -EFAULT;
52                  ptr += sizeof(tr);
53                  ......
54
55                  list_del(&t->work.entry);
56                  t->buffer->allow_user_free = 1;
57                  if (cmd == BR_TRANSACTION && !(t->flags & TF_ONE_WAY)) {
58                          ......
59                  } else {
60                          t->buffer->transaction = NULL;
61                          kfree(t);
62                          ......
63                  }
64                  break;
65          }
66
67  done:
68          *consumed = ptr - buffer;
69          ......
70          return 0;
71  }
```

第16行首先从线程thread的todo队列中取出这个类型为BINDER_WORK_TRANSACTION的工作项，并且保存在变量w中。接着第27行又将该工作项的宿主binder_transaction结构体取回来，并且保存在变量t中。

当一个Server线程将一个进程间通信结果返回给一个Client线程时，是不需要在进程间通信结果数据中指定一个目标Binder实体对象的，因此，前面得到的binder_transaction结构体t所使用的内核缓冲区buffer的成员变量target_node的值就等于NULL。这时候函数binder_thread_read就知道需要向线程thread发送一个BR_REPLY返回协议了，即第38行将变量cmd的值设置为BR_REPLY，以便可以将进程间通信结果返回给它。

返回给线程thread的进程间通信结果数据保存在binder_transaction结构体t的内核缓冲区buffer中，因此，第42行到第45行代码就将它们复制到binder_transaction_data结构体tr中。接着第47行和第50行又分别将BR_REPLY返回协议代码和binder_transaction_data结构体tr的内容，复制到由线程thread提供的一个用户空间缓冲区中。

现在，从线程thread的todo队列中取出来的工作项w已经得到处理了，因此，第55行就将它从队列中删除。接着第56行又将binder_transaction结构体t的内核缓冲区buffer的成员变量allow_user_free的值设置为1，表示该内核缓冲区可以由用户空间通过BC_FREE_BUFFER命令协议来释放。

与BR_TRANSACTION返回协议的处理过程不同，Binder驱动程序向目标线程发送一个BR_REPLY返回协议之后，它就可以将与该BR_REPLY返回协议所关联的binder_transaction结构体释放了，因为Binder驱动程序不需要等待目标线程将BR_REPLY返回协议的处理结果返回来。因此，第61行就调用函数kfree来释放binder_transaction结构体t所占用的内存。

注意

这时候分配给binder_transaction结构体t的内核缓冲区buffer还未被释放，因为Binder驱动程序还需要通过它将进程间通信结果传递给线程thread。当线程thread处理完成Binder驱动程序给它发送的BR_REPLY返回协议之后，它就会使用BC_FREE_BUFFER命令协议来释放分配给binder_transaction结构体t的内核缓冲区buffer。

线程thread执行完成函数binder_thread_read之后，先返回到函数binder_ioctl中，接着再返回到IPCThreadState类的成员函数talkWithDriver中，最后返回到IPCThreadState类的成员函数waitForResponse中来处理BR_REPLY返回协议，如下所示。

frameworks/base/libs/binder/IPCThreadState.cpp

```
01 status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
02 {
03     int32_t cmd;
04     int32_t err;
05
06     while (1) {
07         if ((err=talkWithDriver()) < NO_ERROR) break;
08         ......
09
10         cmd = mIn.readInt32();
11
12         ......
13
14         switch (cmd) {
15         ......
16         case BR_REPLY:
17             {
18                 binder_transaction_data tr;
19                 err = mIn.read(&tr, sizeof(tr));
20                 ......
21
22                 if (reply) {
23                     if ((tr.flags & TF_STATUS_CODE) == 0) {
24                         reply->ipcSetDataReference(
25                             reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
26                             tr.data_size,
27                             reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
28                             tr.offsets_size/sizeof(size_t),
29                             freeBuffer, this);
30                     } else {
31                         ......
32                     }
33                 } else {
34                     ......
35                 }
36             }
37             goto finish;
38         ......
39         }
40
41     }
42
43 finish:
44     ......
45
46     return err;
47 }
```

IPCThreadState类的成员函数talkWithDriver在返回到成员函数waitForResponse中之前，已经将从

Binder驱动程序中读取回来的BR_REPLY返回协议的内容保存在IPCThreadState类内部的返回协议缓冲区mIn中，因此，第10行代码就可以首先从返回协议缓冲区mIn中读出一个BR_REPLY返回协议代码。由于BR_REPLY返回协议的内容是使用一个binder_transaction_data结构体来描述的，因此，第19行代码就可以将返回协议缓冲区mIn的内容读到一个binder_transaction_data结构体tr中。

从前面5.8.1.4小节的内容可以知道，如果binder_transaction_data结构体tr的成员变量flags的TF_STATUS_CODE位等于0，即第23行的if语句为true，就说明当前线程之前所发出的一个进程间通信请求已经被成功地处理了。因此，第24行到第29行代码就将保存在binder_transaction_data结构体tr中的进程间通信结果保存在Parcel对象reply中，这是通过调用Parcel对象reply的成员函数ipcSetDataReference来实现的。

Parcel类的成员函数ipcSetDataReference的实现如下所示。

frameworks/base/libs/binder/Parcel.cpp

```
01 void Parcel::ipcSetDataReference(const uint8_t* data, size_t dataSize,
02     const size_t* objects, size_t objectsCount, release_func relFunc, void* relCookie)
03 {
04     freeDataNoInit();
05     mError = NO_ERROR;
06     mData = const_cast<uint8_t*>(data);
07     mDataSize = mDataCapacity = dataSize;
08     ......
09     mDataPos = 0;
10     ......
11     mObjects = const_cast<size_t*>(objects);
12     mObjectsSize = mObjectsCapacity = objectsCount;
13     ......
14     mOwner = relFunc;
15     mOwnerCookie = relCookie;
16     ......
17 }
```

第4行首先调用函数freeDataNoInit来释放当前Parcel对象内部的数据缓冲区所占用的内存，接着第5行到第15行代码再重新初始化当前Parcel对象内部所使用的数据缓冲区。

参数data指向了用来保存进程间通信结果数据的缓冲区，它的大小由参数dataSize来描述，第6行到第9行代码将该缓冲区作为当前Parcel对象内部使用的数据缓冲区。

参数objects指向了一个Binder对象偏移数组，用来描述保存在进程间通信数据缓冲区中的Binder对象的位置，它的大小由参数objectsCount来描述，第11行和第12行代码将该偏移数组作为当前Parcel对象内部使用的偏移数组。

参数relFunc是一个函数指针，它指向了IPCThreadState类的成员函数freeBuffer；参数relCookie指向了当前线程的IPCThreadState对象，第14行和第15行代码分别将它们保存在当前Parcel对象的成员变量mOwner和mOwnerCookie中。线程获得了保存在当前Parcel对象中的进程间通信结果数据之后，它就会析构该Parcel对象，而该Parcel对象在析构时，会通过它的成员变量mOwner和mOwnerCookie来调用IPCThreadState类的成员函数freeBuffer释放它内部使用的数据缓冲区mData。由于这个数据缓冲区mData是在Binder驱动程序中分配的，即它指向的是一个内核缓冲区，因此，IPCThreadState类的成员函数freeBuffer会通过BC_FREE_BUFFER命令协议通知Binder驱动程序将该内核缓冲区释放掉。

返回到IPCThreadState类的成员函数waitForResponse中，然后再沿着调用过程，最后返回到FregServer进程的main函数中，至此，Service组件FregService的注册过程就执行完成了。

注册了Service组件FregService之后，接下来FregServer进程就要启动一个Binder线程池来等待Client进程向Service组件FregService发送进程间通信请求。

5.8.2 启动Binder线程池

一个进程是通过调用其内部的ProcessState对象的成员函数startThreadPool来启动一个Binder线程池的。例如，FregServer进程将Service组件FregService注册到Service Manager中之后，启动Binder线程池的代码片段如下所示。

external/binder/server/FregServer.cpp

```
1 int main(int argc, char** argv)
2 {
3         ......
4
5         ProcessState::self()->startThreadPool();
6         IPCThreadState::self()->joinThreadPool();
7
8         return 0;
9 }
```

第5行调用当前进程的ProcessState对象的成员函数startThreadPool来启动一个Binder线程池，接着第6行继续调用当前线程的IPCThreadState对象的成员函数joinThreadPool，将当前线程加入到前面所启动的Binder线程池中去等待和处理来自Client进程的进程间通信请求。

下面我们就分析ProcessState类的成员函数startThreadPool的实现，在分析过程中，同时也会分析IPCThreadState类的成员函数joinThreadPool的实现。

ProcessState类的成员函数startThreadPool的实现如下所示。

frameworks/base/libs/binder/ProcessState.cpp

```
1 void ProcessState::startThreadPool()
2 {
3     AutoMutex _l(mLock);
4     if (!mThreadPoolStarted) {
5         mThreadPoolStarted = true;
6         spawnPooledThread(true);
7     }
8 }
```

当前进程的ProcessState对象的成员变量mThreadPoolStarted被初始化为false，当它将一个Binder线程池启动起来之后，就会将内部的成员变量mThreadPoolStarted的值设置为true，防止它的成员函数spawnPooledThread被重复调用来启动Binder线程池。

ProcessState类的成员函数spawnPooledThread的实现如下所示。

frameworks/base/libs/binder/ProcessState.cpp

```
01 void ProcessState::spawnPooledThread(bool isMain)
02 {
03     if (mThreadPoolStarted) {
04         int32_t s = android_atomic_add(1, &mThreadPoolSeq);
05         char buf[32];
06         sprintf(buf, "Binder Thread #%d", s);
07         LOGV("Spawning new pooled thread, name=%s\n", buf);
08         sp<Thread> t = new PoolThread(isMain);
09         t->run(buf);
10     }
11 }
```

第8行创建了一个PoolThread对象t，接着第9行调用它的成员函数run来启动一个新的线程。

注意

前面传进来的参数isMain的值为true，表示线程t是进程主动创建来加入到它的Binder线程池的，以区别于Binder驱动程序请求进程创建新的线程来加入到它的Binder线程池的情况。

PoolThread类继承了线程类Thread，并且重写了它的线程入口成员函数threadLoop，因此，当一个PoolThread对象t所对应的线程启动起来之后，它的成员函数threadLoop就会被调用。

PoolThread类的成员函数threadLoop的实现如下所示。

frameworks/base/libs/binder/ProcessState.cpp

```
01 class PoolThread : public Thread
02 {
03 ......
04 protected:
05     virtual bool threadLoop()
06     {
07         IPCThreadState::self()->joinThreadPool(mIsMain);
08         return false;
09     }
10
11     const bool mIsMain;
12 };
```

第7行调用当前线程中的IPCThreadState对象的成员函数joinThreadPool，将当前线程注册到Binder驱动程序中去成为一个Binder线程，以便Binder驱动程序可以分发进程间通信请求给它处理。

IPCThreadState类的成员函数joinThreadPool的实现如下所示。

frameworks/base/libs/binder/IPCThreadState.cpp

```
01 void IPCThreadState::joinThreadPool(bool isMain)
02 {
03     ......
04
05     mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);
06
07     ......
08
09     status_t result;
10     do {
11         int32_t cmd;
12
13         ......
14
15         // now get the next command to be processed, waiting if necessary
16         result = talkWithDriver();
17         if (result >= NO_ERROR) {
18             size_t IN = mIn.dataAvail();
19             if (IN < sizeof(int32_t)) continue;
20             cmd = mIn.readInt32();
21             ......
22
23             result = executeCommand(cmd);
24         }
25
26         ......
27
28         // Let this thread exit the thread pool if it is no longer
29         // needed and it is not the main process thread.
30         if(result == TIMED_OUT && !isMain) {
31             break;
32         }
33     } while (result != -ECONNREFUSED && result != -EBADF);
34
35     ......
36
37     mOut.writeInt32(BC_EXIT_LOOPER);
38     talkWithDriver(false);
39 }
```

参数isMain是一个默认参数，它的默认值为true。从前面的调用过程可以知道，无论是FregServer进程的主线程，还是FregServer进程刚才所创建的线程，它们都是主动请求加入到Binder线程池的，即它们都不是由于Binder驱动程序请求创建而加入到Binder线程池的，因此，当它们执行到这里的时候，传进来的参数isMain的值均等于true。

一个Binder线程的生命周期可以划分为三个阶段：第一阶段是将自己注册到Binder线程池中；第二个阶段是在一个无限循环中不断地等待和处理进程间通信请求；第三个阶段是退出Binder线程池。

第5行代码根据参数isMain的值不同，相应地将一个BC_ENTER_LOOPER或者BC_REGISTER_LOOPER命令协议写入到IPCThreadState类内部的命令协议缓冲区mOut中，以便接下来可以调用成员函数talkWithDriver将它里面的BC_ENTER_LOOPER或者BC_REGISTER_LOOPER命令协议发送给Binder驱动程序，从而完成注册Binder线程的过程。

第10行到第33行的while循环不断地调用IPCThreadState类的成员函数talkWithDriver来和Binder驱动程序交互，这是通过IO控制命令BINDER_WRITE_READ实现的。在进入Binder驱动程序之前，IPCThreadState类的成员函数talkWithDriver会检查内部的命令协议缓冲区mOut是否有命令协议，如果有，就会将它们发送给Binder驱动程序处理。从Binder驱动程序返回之后，如果Binder驱动程序给它分发了返回协议，那么IPCThreadState类的成员函数talkWithDriver就会将它们保存在内部的返回协议缓冲区mIn中，接着第23行调用成员函数executeCommand来处理它们。

如果IPCThreadState类的成员函数talkWithDriver长时间没有等到进程间通信请求，或者成员函数executeCommand在处理一个进程间通信请求时发生超时，即变量result的值等于TIMED_OUT，并且当前线程不是进程主动创建来处理进程间通信请求的，即参数isMain 的值等于false，那么第31行就跳出第10行到第33行的while循环。最后，当前线程就通过第37行和第38行代码向Binder驱动程序发送一个BC_EXIT_LOOPER命令协议，通知Binder驱动程序它要退出Binder线程池了。

至此，一个Service组件的启动过程就分析完成了。在接下来的5.9小节中，我们继续分析Client进程是如何获得一个已经注册到Service Manager中的Service组件的代理对象的。

5.9　Service代理对象的获取过程

Service组件将自己注册到Service Manager中之后，它就在Server进程中等待Client进程将进程间通信请求发送过来。Client进程为了和运行在Server进程中的Service组件通信，首先要获得它的一个代理对象，这是通过Service Manager提供的Service组件查询服务来实现的。在本节中，我们将以5.3小节中介绍的应用程序FregClient为例，详细分析一个Service代理对象的获取过程。

FregService代理对象的类型为BpFregService，它用来描述一个实现了IFregService接口的Client组件。根据前面5.2小节中的图5-16所描述的Client组件实现原理图，我们将图中的IXXXXXX接口和BpXXXXXX类直接替换为IFregService接口和BpFregService类，就可以得到FregService代理对象的类关系图，如图5-24所示。

为了创建一个FregService代理对象，即一个BpFregService对象，首先要通过Binder驱动程序来获得一个引用了运行在FregServer进程中的FregService组件的Binder引用对象的句柄值，然后再通过这个句柄值创建一个Binder代理对象，即一个BpBinder对象，最后将这个Binder代理对象封装成一个FregService代理对象。

Service Manager代理对象的成员函数getService提供了获取一个Service组件的代理对象的功能，而Service Manager代理对象可以通过Binder库提供的函数defaultServiceManager 来获得。在调用Service Manager代理对象的成员函数getService来获得一个Service组件的代理对象时，需要指定这个Service组件注册到Service Manager中的名称。

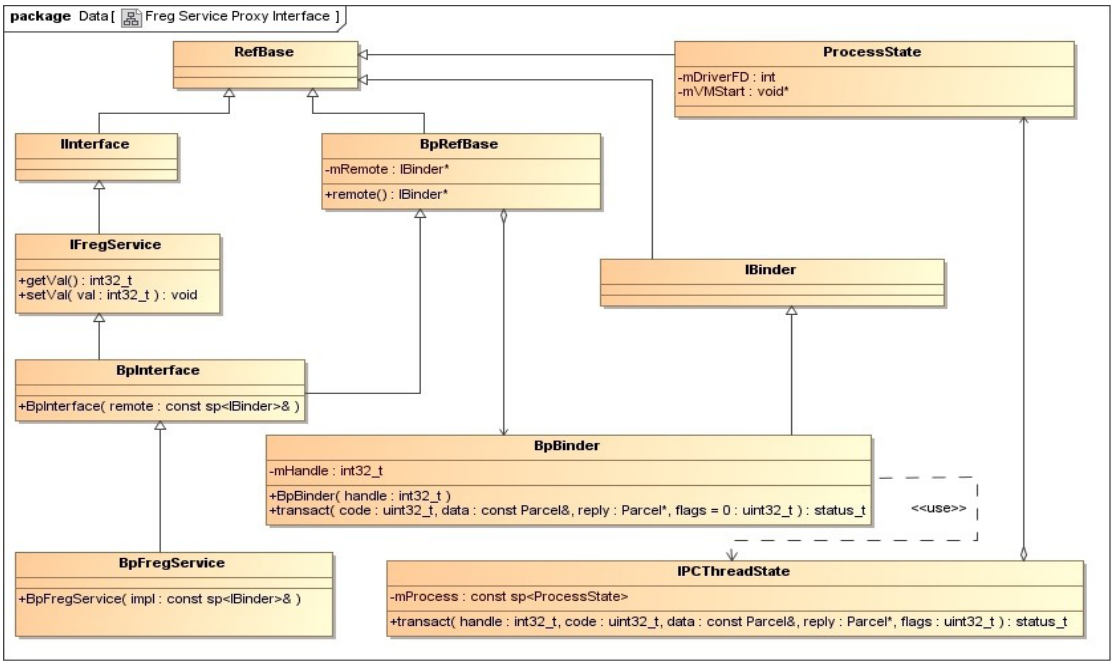


图5-24　FregService代理对象的类关系图

在前面的5.3小节中，FregClient进程获得运行在FregServer进程中的Service组件FregService的代理对象的代码片段如下所示。

external/binder/client/FregClient.cpp

```
01 int main()
02 {
03         sp<IBinder> binder = defaultServiceManager()->getService(String16(FREG_SERVICE));
04         if(binder == NULL) {
05                 LOGE("Failed to get freg service: %s.\n", FREG_SERVICE);
06                 return -1;
07         }
08
09         sp<IFregService> service = IFregService::asInterface(binder);
10         if(service == NULL) {
11                 LOGE("Failed to get freg service interface.\n");
12                 return -2;
13         }
14
15           ......
16 }
```

第3行首先调用Binder库提供的函数defaultServiceManager在FregClient进程中获得一个Service Manager代理对象，接着再调用它的成员函数getService来获得一个名称为FREG_SERVICE的Service组件的代理对象。由于前面获得的Service组件的代理对象的类型为BpBinder，因此，第9行需要通过IFregService类的静态成员函数asInterface将它封装成一个BpFregService类型的代理对象，并且获得它的一个IFregService接口，最后就可以通过这个接口来向运行在FregServer进程中的Service组件FregService发送进程间通信请求了。

接下来，我们就分别分析Service Manager代理对象的成员函数getService和IFregService类的静态成员函数asInterface的实现。

Service Manager代理对象的成员函数getService的实现如下所示。

frameworks/base/libs/binder/IServiceManager.cpp

```
01 class BpServiceManager : public BpInterface<IServiceManager>
02 {
03 public:
04     ......
05
06     virtual sp<IBinder> getService(const String16& name) const
07     {
08         unsigned n;
09         for (n = 0; n < 5; n++){
10             sp<IBinder> svc = checkService(name);
11             if (svc != NULL) return svc;
12             LOGI("Waiting for service %s...\n", String8(name).string());
13             sleep(1);
14         }
15         return NULL;
16     }
17
18     ......
19 };
```

这个函数最多会尝试5次来获得一个名称为name的Service组件的代理对象。如果上一次获得失败，那么第13行就调用函数sleep使得当前线程睡眠1毫秒，然后再重新去获取；否则，第11行就直接将获得的Service组件的代理对象返回给调用者。

第10行调用了成员函数checkService来获得一个名称为name的Service组件的代理对象，它的实现如下所示。

frameworks/base/libs/binder/IServiceManager.cpp

```
01 class BpServiceManager : public BpInterface<IServiceManager>
02 {
03 public:
04     ......
05
06     virtual sp<IBinder> checkService( const String16& name) const
07     {
08         Parcel data, reply;
09         data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
10         data.writeString16(name);
11         remote()->transact(CHECK_SERVICE_TRANSACTION, data, &reply);
12         return reply.readStrongBinder();
13     }
14
15     ......
16 };
```

与在前面5.8小节中分析的Service Manager代理对象的成员函数addService的实现类似，Service Manager代理对象的成员函数getService也是通过内部一个句柄值为0的Binder代理对象来与Service Manager通信的。两者的区别在于请求Service Manager执行的操作不同，其中，前者请求Service Manager执行一个ADD_SERVICE_TRANSACTION操作，而后者请求Service Manager执行一个CHECK_SERVICE_TRANSACTION操作。

Service Manager代理对象的成员函数getService实现的是一个标准的Binder进程间通信过程，它可以划分为下面五个步骤。

（1）FregClient进程将进程间通信数据，即要获得其代理对象的Service组件FregService的名称，封装在一个Parcel对象中，用来传递给Binder驱动程序。

（2）FregClient进程向Binder驱动程序发送一个BC_TRANSACTION命令协议，Binder驱动程序根据协议内容找到Service Manager进程之后，就会向FregClient进程发送一个BR_TRANSACTION_

COMPLETE返回协议，表示它的进程间通信请求已经被接受。FregClient进程接收到Binder驱动程序发送给它的BR_TRANSACTION_COMPLETE返回协议，并且对它进行处理之后，就会再次进入到Binder驱动程序中去等待Service Manager进程将它要获取的Binder代理对象的句柄值返回来。

（3）Binder驱动程序在向FregClient进程发送BR_TRANSACTION_COMPLETE返回协议的同时，也会向Service Manager进程发送一个BR_TRANSACTION返回协议，请求Service Manager进程执行一个CHECK_SERVICE_TRANSACTION操作。

（4）Service Manager进程执行完成FregClient进程请求的CHECK_SERVICE_TRANSACTION操作之后，就会向Binder驱动程序发送一个BC_REPLY命令协议，协议内容包含了Service组件FregService的信息。Binder驱动程序根据协议内容中的Service组件FregService的信息为FregClient进程创建一个Binder引用对象，接着就会向Service Manager进程发送一个BR_TRANSACTION_COMPLETE返回协议，表示它返回的Service组件FregService的信息已经收到了。Service Manager进程接收到Binder驱动程序发送给它的BR_TRANSACTION_COMPLETE返回协议，并且对它进行处理之后，一次进程间通信过程就结束了，接着它会再次进入到Binder驱动程序中去等待下一次进程间通信请求。

（5）Binder驱动程序在向Service Manager进程发送BR_TRANSACTION_COMPLETE返回协议的同时，也向FregClient进程发送一个BR_REPLY返回协议，协议内容包含了前面所创建的一个Binder引用对象的句柄值，这时候FregClient进程就可以通过这个句柄来创建一个Binder代理对象。

由于这是一个标准的Binder进程间通信过程，并且我们已经在前面的5.8.1小节中详细分析过这五个步骤了，因此，在介绍获取Service组件FregSerivce的代理对象的过程中，我们只分析以下三个子过程。

（1）Service Manager进程处理操作代码为CHECK_SERVICE_TRANSACTION的进程间通信请求的过程。

（2）Binder驱动程序为FregClient进程创建一个引用了Service组件FregService的Binder引用对象的过程。

（3）Binder库为FregClient进程创建一个Binder代理对象的过程。

从前面5.8.1.3的内容可以知道，Service Manager是统一在函数svcmgr_handler中处理来自Client进程的进程间通信请求的，它处理操作代码为CHECK_SERVICE_TRANSACTION的进程间通信请求的过程如下所示。

frameworks/base/cmds/servicemanager/service_manager.c

```
01 int svcmgr_handler(struct binder_state *bs,
02                     struct binder_txn *txn,
03                     struct binder_io *msg,
04                     struct binder_io *reply)
05 {
06     struct svcinfo *si;
07     uint16_t *s;
08     unsigned len;
09     void *ptr;
10     ......
11
12     switch(txn->code) {
13     case SVC_MGR_GET_SERVICE:
14     case SVC_MGR_CHECK_SERVICE:
15         s = bio_get_string16(msg, &len);
16         ptr = do_find_service(bs, s, len);
17         if (!ptr)
18             break;
19         bio_put_ref(reply, ptr);
20         return 0;
21     ......
```

```
22      }
23
24      ......
25  }
```

FregClient进程传递过来的Service组件FregService的名称为FREG_SERVICE，即“shy.luo.FregService”，第15行调用函数bio_get_string16将它从binder_io结构体msg的数据缓冲区中获取回来，接着第16行调用函数do_find_service在已注册Service组件列表svclist中查找与它对应的一个svcinfo结构体。

函数do_find_service的实现如下所示。

frameworks/base/cmds/servicemanager/service_manager.c

```
01 void *do_find_service(struct binder_state *bs, uint16_t *s, unsigned len)
02 {
03      struct svcinfo *si;
04      si = find_svc(s, len);
05
06      ......
07      if (si && si->ptr) {
08          return si->ptr;
09      } else {
10          return 0;
11      }
12 }
```

第4行调用函数find_svc来查找与字符串s对应的一个svcinfo结构体si。在前面的5.8.1.3小节中，我们已经分析过函数find_svc的实现了，它通过遍历已注册Service组件列表svclist来查找与字符串s对应的一个svcinfo结构体。如果找到了与字符串s对应的svcinfo结构体si，并且它的成员变量ptr的值不为0，即第7行的if语句为true，那么第8行就将它的成员变量ptr的值返回给调用者。

从前面5.8.1.3小节的内容可以知道，结构体svcinfo的成员变量ptr保存的是一个引用了注册到Service Manager中的Service组件的Binder引用对象的句柄值。当Service Manager将这个句柄值返回给Binder驱动程序时，Binder驱动程序就可以根据它找到相应的Binder引用对象，接着找到该Binder引用对象所引用的Binder实体对象，最后Binder驱动程序就可以在请求获取该Service组件的代理对象的Client进程中创建另一个Binder引用对象了。

返回到函数svcmgr_handler中，接着第19行将前面获得的一个句柄值封装成一个binder_object结构体，并且写入到binder_io结构体reply中，以便可以将它返回给Binder驱动程序，这是通过调用函数bio_put_ref来实现的。

函数bio_put_ref的实现如下所示。

frameworks/base/cmds/servicemanager/binder.c

```
01 void bio_put_ref(struct binder_io *bio, void *ptr)
02 {
03      struct binder_object *obj;
04
05      if (ptr)
06          obj = bio_alloc_obj(bio);
07      else
08          obj = bio_alloc(bio, sizeof(*obj));
09
10      if (!obj)
11          return;
12
13      obj->flags = 0x7f | FLAT_BINDER_FLAG_ACCEPTS_FDS;
14      obj->type = BINDER_TYPE_HANDLE;
15      obj->pointer = ptr;
16      obj->cookie = 0;
17 }
```

由于传进来的参数ptr的值不等于0，因此，第6行就调用函数bio_alloc_obj在binder_io结构体bio的数据缓冲区中分配一个binder_object结构体obj。

函数bio_alloc_obj的实现如下所示。

frameworks/base/cmds/servicemanager/binder.c

```
01 static struct binder_object *bio_alloc_obj(struct binder_io *bio)
02 {
03     struct binder_object *obj;
04
05     obj = bio_alloc(bio, sizeof(*obj));
06
07     if (obj && bio->offs_avail) {
08         bio->offs_avail--;
09         *bio->offs++ = ((char*) obj) - ((char*) bio->data0);
10         return obj;
11     }
12
13     bio->flags |= BIO_F_OVERFLOW;
14     return 0;
15 }
```

第5行调用函数bio_alloc在binder_io结构体bio的数据缓冲区中分配一个未初始化的binder_object结构体obj，接着第7行到第11行代码在binder_io结构体bio的偏移数组中分配一个元素来保存binder_object结构体obj在数据缓冲区中的位置。这样Binder驱动程序就可以知道Service Manager给它返回的进程间通信结果数据中包含了一个Binder对象。

回到函数bio_put_ref中，第13行到第16行代码对前面从binder_io结构体bio的数据缓冲区中分配的binder_object结构体obj进行初始化。第14行将binder_object结构体obj所描述的Binder对象的类型设置为BINDER_TYPE_HANDLE，接着第15行又将这个Binder对象的句柄值设置为ptr。

回到函数svcmgr_handler中，现在要返回给Binder驱动程序的进程间通信结果数据都保存在binder_io结构体reply中了。接着它又返回到函数binder_parse中，最后调用函数binder_send_reply将binder_io结构体reply的内容返回给Binder驱动程序，这是通过向Binder驱动程序发送一个BC_REPLY命令协议来实现的。

在前面的5.8.1.4小节中，我们已经分析过函数binder_send_reply向Binder驱动程序发送一个BC_REPLY命令协议的过程了。Binder驱动程序接收到Service Manager发送给它的BC_REPLY命令协议之后，就在函数binder_transaction中对它进行处理，如下所示。

kernel/goldfish/drivers/staging/android/binder.c

```
01 static void
02 binder_transaction(struct binder_proc *proc, struct binder_thread *thread,
03         struct binder_transaction_data *tr, int reply)
04 {
05         struct binder_transaction *t;
06         struct binder_work *tcomplete;
07         size_t *offp, *off_end;
08         struct binder_proc *target_proc;
09         struct binder_thread *target_thread = NULL;
10         struct binder_node *target_node = NULL;
11         struct list_head *target_list;
12         wait_queue_head_t *target_wait;
13         struct binder_transaction *in_reply_to = NULL;
14         ......
15
16         t = kzalloc(sizeof(*t), GFP_KERNEL);
17         ......
18
```

```
19          tcomplete = kzalloc(sizeof(*tcomplete), GFP_KERNEL);
20          ......
21
22          offp = (size_t *)(t->buffer->data + ALIGN(tr->data_size, sizeof(void *)));
23
24          ......
25
26          off_end = (void *)offp + tr->offsets_size;
27
28          for (; offp < off_end; offp++) {
29                  struct flat_binder_object *fp;
30                  ......
31                  fp = (struct flat_binder_object *)(t->buffer->data + *offp);
32                  switch (fp->type) {
33                  ......
34                  case BINDER_TYPE_HANDLE:
35                  case BINDER_TYPE_WEAK_HANDLE: {
36                          struct binder_ref *ref = binder_get_ref(proc, fp->handle);
37                          ......
38                          if (ref->node->proc == target_proc) {
39                                  ......
40                          } else {
41                                  struct binder_ref *new_ref;
42                                  new_ref = binder_get_ref_for_node(target_proc, ref->node);
43                                  ......
44                                  fp->handle = new_ref->desc;
45                                  binder_inc_ref(new_ref, fp->type == BINDER_TYPE_HANDLE, NULL);
46                                  ......
47                          }
48                  } break;
49                  ......
50                  }
51          }
52
53          t->work.type = BINDER_WORK_TRANSACTION;
54          list_add_tail(&t->work.entry, target_list);
55          tcomplete->type = BINDER_WORK_TRANSACTION_COMPLETE;
56          list_add_tail(&tcomplete->entry, &thread->todo);
57          if (target_wait)
58                  wake_up_interruptible(target_wait);
59          return;
60
61          ......
62  }
```

在前面的5.8.1.4小节中，我们已经分析过函数binder_transaction处理BC_REPLY命令协议的过程了，这里只关注它处理进程间通信结果数据中的Binder对象的过程，即第28行到第51行的for循环的处理逻辑。

从前面的调用过程可以知道，Service Manager进程返回给Binder驱动程序的进程间通信结果数据中包含了一个类型为BINDER_TYPE_HANDLE的binder_object结构体，即一个flat_binder_object结构体。因此，第36行到第47行代码就需要对这个flat_binder_object结构体进行处理。

第36行根据flat_binder_object结构体fp的句柄值在进程proc中找到对应的Binder引用对象ref。由于Binder引用对象ref所引用的Binder本地对象是运行在FregServer进程中的Service组件FregService，而目标进程target_proc指向了要获取Service组件FregService的代理对象的FregClient进程，因此，第38行的if语句为false，接下来就会调用第41行到第45行代码。

第42行调用函数binder_get_ref_for_node在FregClient进程中查找是否已经存在一个Binder引用对象，它引用了Binder实体对象ref->node，即引用了Service组件FregService。如果不存在，那么函数

binder_get_ref_for_node就会为FregClient进程创建一个引用了Service组件FregService的Binder引用对象，并且返回给调用者；否则，就直接将该Binder引用对象返回给调用者。

第44行将flat_binder_object结构体fp的句柄值修改为FregClient进程创建的一个Binder引用对象new_ref的句柄值。因为接下来Binder驱动程序需要将flat_binder_object结构体fp返回给FregClient进程，以便FregClient进程可以根据这个句柄值创建一个引用了Service组件FregService的Binder代理对象。

第45行调用函数binder_inc_ref增加Binder引用对象new_ref的引用计数，因为它被保存在binder_transaction结构体t的内核缓冲区buffer中，即被binder_transaction结构体t的内核缓冲区buffer引用了。等到binder_transaction结构体t的内核缓冲区buffer被释放时，Binder驱动程序就会相应地减少Binder引用对象new_ref的引用计数。

第53行和第54行将binder_transaction结构体t封装成一个类型为BINDER_WORK_TRANSACTION的工作项添加到目标进程target_proc或者目标线程target_thread的todo队列中，并且第58行将目标进程target_proc或者目标线程target_thread唤醒，以便它们可以处理这个工作项。

第55行和第56行将binder_work结构体tcomplete封装成一个类型为BINDER_WORK_TRANSACTION_COMPLETE的工作项添加到源线程thread的todo队列中，以便它从Binder驱动程序返回到用户空间之前，可以处理该工作项。

接下来，我们假设类型为BINDER_WORK_TRANSACTION的工作项是添加到目标线程target_thread的todo队列中的，即添加到调用Service Manager代理对象的成员函数getService来获得一个Service组件FregService的代理对象的FregClient线程的todo队列中。当这个目标线程target_thread被唤醒之后，它就会继续执行Binder驱动程序中的函数binder_thread_read，最终获得了一个BR_REPLY返回协议，协议内容包含了前面为FregClient进程创建的Binder引用对象new_ref的句柄值。

目标线程target_thread返回到用户空间之后，就在IPCThreadState类的成员函数waitForResponse中处理从Binder驱动程序中读取回来的BR_REPLY返回协议，它主要是将协议内容，即进程间通信结果数据封装在一个Parcel对象reply中，最后返回到Service Manager代理对象的成员函数checkService中。

为了方便描述，我们将Service Manager代理对象的成员函数checkService的代码重新列出来，如下所示。

frameworks/base/libs/binder/IServiceManager.cpp

```
01 class BpServiceManager : public BpInterface<IServiceManager>
02 {
03 public:
04     ......
05
06     virtual sp<IBinder> checkService( const String16& name) const
07     {
08         Parcel data, reply;
09         data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
10         data.writeString16(name);
11         remote()->transact(CHECK_SERVICE_TRANSACTION, data, &reply);
12         return reply.readStrongBinder();
13     }
14
15     ......
16 }
```

当前线程从IPCThreadState类的成员函数waitForResponse返回到Service Manager代理对象的成员函数checkService之后，第12行就调用Parcel对象reply的成员函数readStrongBinder来获得一个Binder代理对象。

Parcel类的成员函数readStrongBinder的实现如下所示。

frameworks/base/libs/binder/Parcel.cp

```
1 sp<IBinder> Parcel::readStrongBinder() const
2 {
3     sp<IBinder> val;
4     unflatten_binder(ProcessState::self(), *this, &val);
5     return val;
6 }
```

从前面的调用过程可以知道，Parcel对象reply内部的数据缓冲区保存了Binder驱动程序返回给当前线程的进程间通信结果数据，即它里面包含了一个类型为BINDER_TYPE_HANDLE的flat_binder_object结构体，因此，第12行就调用函数unflatten_binder来获得这个flat_binder_object结构体。

函数unflatten_binder的实现如下所示。

frameworks/base/libs/binder/Parcel.cp

```
01 status_t unflatten_binder(const sp<ProcessState>& proc,
02     const Parcel& in, sp<IBinder>* out)
03 {
04     const flat_binder_object* flat = in.readObject(false);
05
06     if (flat) {
07         switch (flat->type) {
08             ......
09             case BINDER_TYPE_HANDLE:
10                 *out = proc->getStrongProxyForHandle(flat->handle);
11                 return finish_unflatten_binder(
12                     static_cast<BpBinder*>(out->get()), *flat, in);
13         }
14     }
15     return BAD_TYPE;
16 }
```

从Parcel对象in获得的flat_binder_object结构体flat的类型为BINDER_TYPE_HANDLE，因此，第10行就调用当前进程的ProcessState对象proc的成员函数getStrongProxyForHandle来查找一个与它的句柄值相对应的Binder代理对象，即一个BpBinder对象。

在前面的5.7小节中介绍Service Manager代理对象的获取过程时，我们已经分析过ProcessState类的成员函数getStrongProxyForHandle的实现了。它首先在内部的一个Binder代理对象列表中检查是否已经存在一个与flat_binder_object结构体flat的句柄值handle相对应的Binder代理对象。如果存在，就将它返回给调用者；否则，就会使用flat_binder_object结构体flat的句柄值handle来创建一个Binder代理对象，然后再返回给调用者。由于flat_binder_object结构体flat的句柄值handle是一个引用了Service组件FregService的Binder引用对象的句柄值，因此，从ProcessState类的成员函数getStrongProxyForHandle返回来的Binder代理对象也是引用了Service组件FregService的。

Service Manager代理对象的成员函数checkService执行完成之后，就将一个Binder代理对象的IBinder接口返回给FregClient进程的入口函数main，接着该函数就继续调用IFregService类的静态成员函数asInterface将该Binder代理对象封装成一个类型为BpFregService的Binder代理对象。

IFregService类的静态成员函数asInterface是通过宏IMPLEMENT_META_INTERFACE来定义的，它的实现如下所示。

```
01 android::sp<IFregService> IFregService::asInterface(
02         const android::sp<android::IBinder>& obj)
03 {
04         android::sp<IFregService> intr;
05         if (obj != NULL) {
06               intr = static_cast<IFregService*>(
07                     obj->queryLocalInterface(
08                           IFregService::descriptor).get());
```

```
09              if (intr == NULL) {
10                    intr = new BpFregService(obj);
11              }
12        }
13        return intr;
14 }
```

参数obj指向的是前面获得的一个Binder代理对象，即一个BpBinder对象，它的成员函数queryLocalInterface的返回值为NULL。因此，第10行将这个Binder代理对象封装成一个BpFregService类型的Binder代理对象，并且将它的IFregService接口返回给调用者。

至此，FregClient进程就成功地通过Service Manager获得了一个运行在FregServer进程中的Service组件FregService的代理对象。有了这个代理对象之后，FregClient进程就可以向Service组件FregService发送进程间通信请求了，即可以使用它提供的服务了。

5.10 Binder进程间通信机制的Java接口

在前面的小节中，我们分析的都是Binder进程间通信机制的C/C++实现，但是Android应用程序是使用Java语言开发的，那么它们是如何使用Binder进程间通信机制的呢？

我们知道，Java代码可以通过JNI方法来调用C/C++代码，因此，Android系统在应用程序框架层中提供了Binder进程间通信机制的Java接口，它们通过JNI方法来调用Binde库的C/C++接口，从而提供了执行Binder进程间通信的能力。

在第2章学习Android系统的硬件抽象层时，我们先开发了一个硬件访问服务FregService，它运行在系统进程System中，并且在启动时通过Java层的Service Manager代理对象将自己注册到Service Manager中。接着又开发了一个应用程序Freg，它通过Java层的Service Manager代理对象来获得硬件访问服务FregService的一个代理对象，最后就可以通过这个代理对象来使用硬件访问服务FregService提供的服务了。

在本节中，我们将结合第2章所介绍的硬件访问服务FregService，以及使用了这个服务的应用程序Freg来分析Binder进程间通信机制的Java接口。

接下来，我们就结合硬件访问服务FregService和应用程序Freg这两个实例，通过下面五个使用情景来分析Binder进程间通信机制的Java接口。

（1）Service Manager的Java代理对象的获取过程。

（2）Java服务接口的定义和解析。

（3）Java服务的启动过程。

（4）Java服务代理对象的获取过程。

（5）Java服务的调用过程。

为了描述方便，在Java层中，我们将Service组件称为Java服务；相应的，将Service组件的代理对象称为Java服务代理对象。学习了Binder进程间通信机制的Java接口之后，我们就可以熟练地在Android应用程序中使用Binder进程间通信机制了。

5.10.1 Service Manager的Java代理对象的获取过程

在Java层中，Service Manager的Java代理对象的类型为ServiceManagerProxy，它实现了IServiceManager

接口，如图5-25所示。

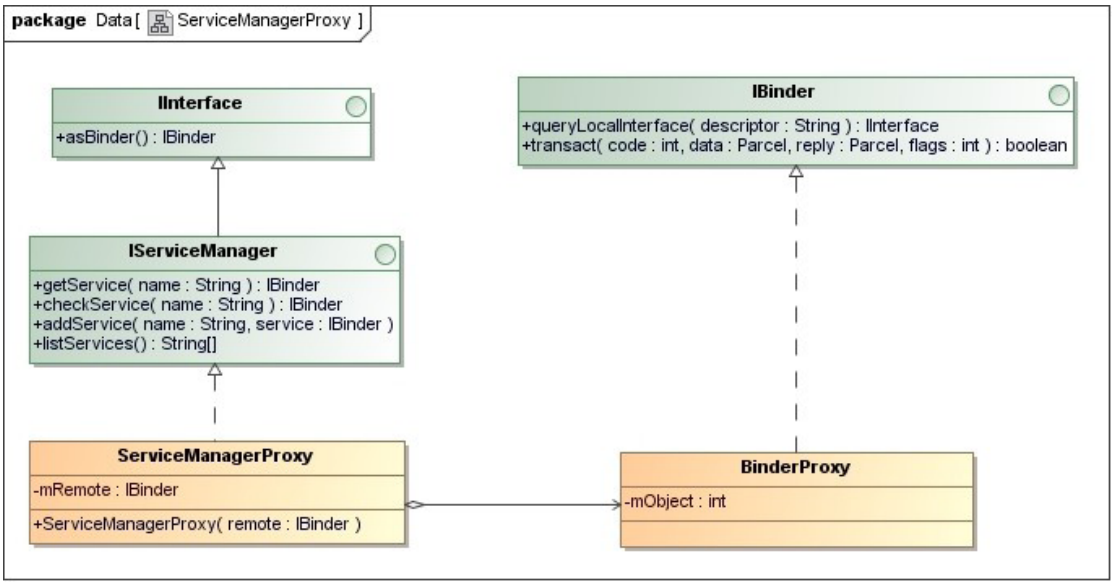


图5-25　Service Manager的Java代理对象的类关系图

IServiceManager接口继承了IInterface接口，并且定义了四个成员函数：getService、checkService、addService和listService，其中，getService和checkService用来获取Java服务的代理对象，addService用来注册Java服务，listService用来获取注册在Service Manager中的Java服务名称列表。

Service Manager的Java代理对象的内部有一个成员变量mRemote，它的类型为IBinder，实际上指向的是一个BinderProxy对象。BinderProxy类用来描述一个Java服务代理对象，它实现了IBinder接口。BinderProxy类内部有一个类型为int的成员变量mObject，它指向C++层中的一个Binder代理对象。这样，我们就可以将一个Java服务代理对象和C++层中的一个Binder代理对象关联起来，即可以通过C++层中的Binder代理对象来实现Java服务代理对象的功能。

从Service Manager的Java代理对象的类关系图就可以知道，在创建一个Service Manager的Java代理对象之前，首先要在C++层中创建一个句柄值为0的Binder代理对象，然后再在Java层创建一个对应的Java服务代理对象。有了这个Java服务代理对象之后，我们就可以创建一个Service Manager的Java代理对象了。

在Java层中，Service Manager的Java代理对象是由ServiceManager类来负责创建的，它的实现如图5-26所示。

ServiceManager类内部有一个类型为ServiceManagerProxy的静态成员变量sServiceManager，ServiceManager类的静态成员函数getService、checkService、addService和listServices就是通过它来实现的。ServiceManager类的另外一个静态成员函数getIServiceManager是用来创建一个Service Manager的Java代理对象的，并且保存在静态成员变量sServiceManager中。ServiceManager类的静态成员函数getIServiceManager在创建一个Service Manager的Java代理对象的过程中，会通过ServiceManagerNative类的静态成员函数asInterface将一个句柄值等于0的Java服务代理对象封装成一个ServiceManagerProxy对象。

Java层的ServiceManagerNative类类似于C++层中的BnServiceManager类，它继承了Java层中的Binder类，是用来实现Java层中的Service Manager服务的。同样， Java层中的Binder类类似于C++层中的BBinder类，是用来实现一个Java服务的。Java层中的Binder类内部有一个类型为int的成员变量mObject，它指向C++层中的一个Binde本地对象。这样，我们就可以将一个Java服务和C++层中的一个Binder本地对象关联起来，即可以通过C++层中的Binder本地对象来实现Java服务的功能。

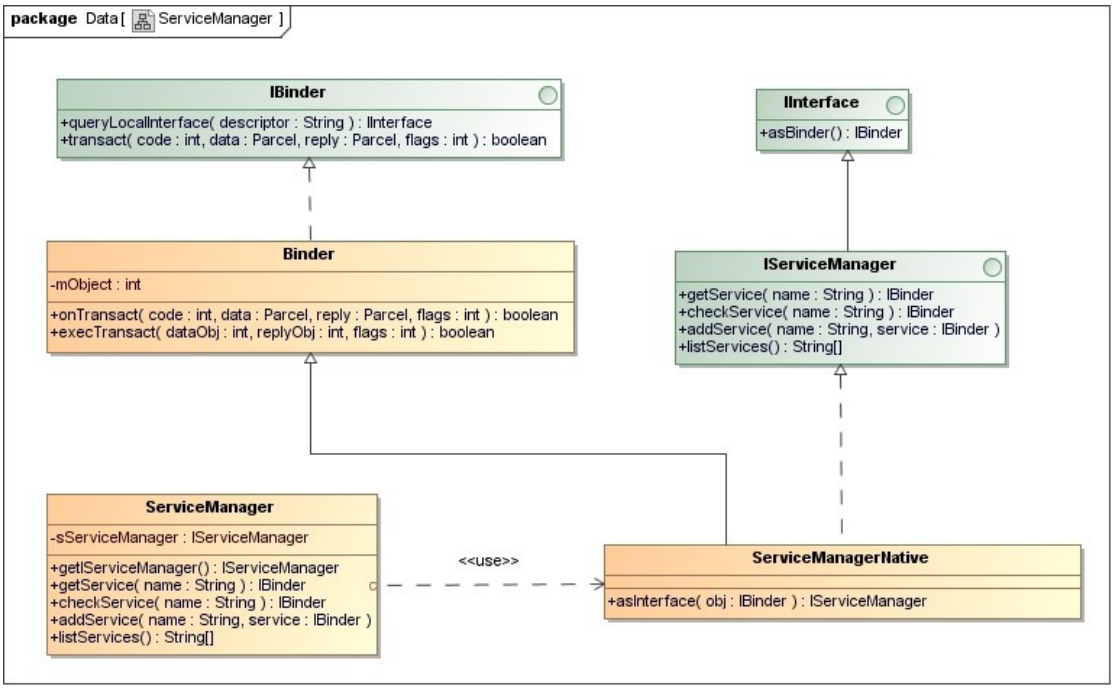


图5-26　Java层的ServiceManager类关系图

接下来，我们就分析ServiceManager类的静态成员函数getIServiceManager是如何创建ServiceManager的Java代理对象的，它的实现如下所示。

frameworks/base/core/java/android/os/ServiceManager.java

```
01 public final class ServiceManager {
02     ......
03     private static IServiceManager sServiceManager;
04     ......
05
06     private static IServiceManager getIServiceManager() {
07         if (sServiceManager != null) {
08             return sServiceManager;
09         }
10
11         // Find the service manager
12         sServiceManager = ServiceManagerNative.asInterface(BinderInternal.getContextObject());
13         return sServiceManager;
14     }
15
16     ......
17 }
```

第7行首先判断ServiceManager类的静态成员变量sServiceManager的值是否不等于null。如果是，那么第8行就直接将它指向的一个ServiceManagerProxy对象返回给调用者；否则，第12行首先通过BinderInternal类的静态成员函数getContextObject创建一个句柄值等于0的Java服务代理对象，接着再通过ServiceManagerNative类的静态成员函数asInterface将这个Java服务代理对象封装成一个ServiceManagerProxy对象，最后保存在静态成员变量sServiceManager中，并且将它返回给调用者。

下面我们分别分析BinderInternal类的静态成员函数getContextObject和ServiceManagerNative类的静态成员函数asInterface的实现。

BinderInternal类的静态成员函数getContextObject是一个JNI方法，它的定义如下所示。

frameworks/base/core/java/com/android/internal/os/BinderInternal.java

```
01 public class BinderInternal {
02     ......
03
04     /**
05      * Return the global "context object" of the system.  This is usually
06      * an implementation of IServiceManager, which you can use to find
07      * other services.
08      */
09     public static final native IBinder getContextObject();
10
11     ......
12 }
```

它是由C++层中的函数android_os_BinderInternal_getContextObject来实现的，如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
1 static jobject android_os_BinderInternal_getContextObject(JNIEnv* env, jobject clazz)
2 {
3     sp<IBinder> b = ProcessState::self()->getContextObject(NULL);
4     return javaObjectForIBinder(env, b);
5 }
```

第3行调用进程中的ProcessState对象的成员函数getContextObject来获得一个句柄值等于NULL，即等于0的Binder代理对象，即一个BpBinder对象，接着第4行调用函数javaObjectForIBinder为这个Binder代理对象创建一个Java服务代理对象，即一个BinderProxy对象，最后将它返回给调用者。

在前面的5.7小节中，我们已经分析过ProcessState类的成员函数getContextObject的实现了。接下来，我们就着重分析函数javaObjectForIBinder的实现，看它是如何创建一个Java服务代理对象的。

在分析函数javaObjectForIBinder的实现之前，我们先介绍三个全局变量：gBinderOffsets、gBinderProxyOffsets和gWeakReferenceOffsets的定义。

全局变量gBinderOffsets是一个bindernative_offsets_t结构体，它的定义如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
01 static struct bindernative_offsets_t
02 {
03     // Class state.
04     jclass mClass;
05     jmethodID mExecTransact;
06
07     // Object state.
08     jfieldID mObject;
09
10 } gBinderOffsets;
```

它有三个成员变量：mClass、mExecTransact和mObject，其中，mClass指向Java层中的Binder类，而mExecTransact和mObject分别指向这个Binder类的成员函数execTransact和成员变量mObject。

全局变量gBinderOffsets是在函数int_register_android_os_Binder中初始化的，如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
01 const char* const kBinderPathName = "android/os/Binder";
02
03 static int int_register_android_os_Binder(JNIEnv* env)
04 {
05     jclass clazz;
06
07     clazz = env->FindClass(kBinderPathName);
08     ......
09
10     gBinderOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
11     gBinderOffsets.mExecTransact
```

```
12          = env->GetMethodID(clazz, "execTransact", "(IIII)Z");
13      ......
14
15      gBinderOffsets.mObject
16          = env->GetFieldID(clazz, "mObject", "I");
17      ......
18 }
```

全局变量gBinderProxyOffsets是一个binderproxy_offsets_t结构体，它的定义如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
01 static struct binderproxy_offsets_t
02 {
03     // Class state.
04     jclass mClass;
05     jmethodID mConstructor;
06     jmethodID mSendDeathNotice;
07
08     // Object state.
09     jfieldID mObject;
10     jfieldID mSelf;
11
12 } gBinderProxyOffsets;
```

它有五个成员变量：mClass、mConstructor、mSendDeathNotice、mObject和mSelf，其中，mClass指向Java层中的BinderProxy类，而mConstructor、mSendDeathNotice、mObject和mSelf 分别指向这个BinderProxy类的构造函数、静态成员函数sendDeathNotice、成员变量mObject和mSelf。

全局变量gBinderProxyOffsets是在函数int_register_android_os_BinderProxy中初始化的，如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
01 const char* const kBinderProxyPathName = "android/os/BinderProxy";
02
03 static int int_register_android_os_BinderProxy(JNIEnv* env)
04 {
05     jclass clazz;
06     ......
07
08     clazz = env->FindClass(kBinderProxyPathName);
09     ......
10
11     gBinderProxyOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
12     gBinderProxyOffsets.mConstructor
13         = env->GetMethodID(clazz, "<init>", "()V");
14     ......
15     gBinderProxyOffsets.mSendDeathNotice
16         = env->GetStaticMethodID(clazz, "sendDeathNotice", "(Landroid/os/IBinder$DeathRecipient;)V");
17     ......
18
19     gBinderProxyOffsets.mObject
20         = env->GetFieldID(clazz, "mObject", "I");
21     ......
22     gBinderProxyOffsets.mSelf
23         = env->GetFieldID(clazz, "mSelf", "Ljava/lang/ref/WeakReference;");
24     ......
25 }
```

全局变量gWeakReferenceOffsets是一个weakreference_offsets_t结构体，它的定义如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
1 static struct weakreference_offsets_t
2 {
3     // Class state.
```

```
4     jclass mClass;
5     jmethodID mGet;
6
7 } gWeakReferenceOffsets;
```

它有两个成员变量：mClass和mGet，其中，mClass指向Java层中的WeakReference类，而mGet指向这个WeakReference类的成员函数get。

全局变量gWeakReferenceOffsets同样是在函数int_register_android_os_BinderProxy中初始化的，如下所示：

frameworks/base/core/jni/android_util_Binder.cpp

```
01 static int int_register_android_os_BinderProxy(JNIEnv* env)
02 {
03     jclass clazz;
04
05     clazz = env->FindClass("java/lang/ref/WeakReference");
06     ......
07     gWeakReferenceOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
08     gWeakReferenceOffsets.mGet
09         = env->GetMethodID(clazz, "get", "()Ljava/lang/Object;");
10     ......
11 }
```

了解了全局变量gBinderOffsets、gBinderProxyOffsets和gWeakReferenceOffsets的定义之后，现在我们就可以分析函数javaObjectForIBinder的实现了，它的定义如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
01 jobject javaObjectForIBinder(JNIEnv* env, const sp<IBinder>& val)
02 {
03     if (val == NULL) return NULL;
04
05     if (val->checkSubclass(&gBinderOffsets)) {
06         // One of our own!
07         jobject object = static_cast<JavaBBinder*>(val.get())->object();
08         ......
09         return object;
10     }
11
12     // For the rest of the function we will hold this lock, to serialize
13     // looking/creation of Java proxies for native Binder proxies.
14     AutoMutex _l(mProxyLock);
15
16     // Someone else's...  do we know about it?
17     jobject object = (jobject)val->findObject(&gBinderProxyOffsets);
18     if (object != NULL) {
19         jobject res = env->CallObjectMethod(object, gWeakReferenceOffsets.mGet);
20         if (res != NULL) {
21             ......
22             return res;
23         }
24         ......
25         val->detachObject(&gBinderProxyOffsets);
26         ......
27     }
28
29     object = env->NewObject(gBinderProxyOffsets.mClass, gBinderProxyOffsets.mConstructor);
30     if (object != NULL) {
31         ......
32         // The proxy holds a reference to the native object.
33         env->SetIntField(object, gBinderProxyOffsets.mObject, (int)val.get());
34         val->incStrong(object);
35
```

```
36          // The native object needs to hold a weak reference back to the
37          // proxy, so we can retrieve the same proxy if it is still active.
38          jobject refObject = env->NewGlobalRef(
39                  env->GetObjectField(object, gBinderProxyOffsets.mSelf));
40          val->attachObject(&gBinderProxyOffsets, refObject,
41                  jnienv_to_javavm(env), proxy_cleanup);
42
43          ......
44      }
45
46      return object;
47  }
```

参数val要么指向一个Binder代理对象，要么指向一个Binder本地对象。当它指向的是一个Binder本地对象时，它的类型为JavaBBinder，后者继承了BBinder类。

第5行调用参数val的成员函数checkSubclass来检查它指向的是一个Binder代理对象，还是一个JavaBBinder对象。

Binder代理对象的成员函数checkSubclass是由其父类IBinder来实现的，如下所示。

frameworks/base/libs/binder/Binder.cpp

```
1 bool IBinder::checkSubclass(const void* /*subclassID*/) const
2 {
3     return false;
4 }
```

它的返回值恒为false。

JavaBBinder对象重写了其父类BBinder的父类IBinder的成员函数checkSubclass，如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
01 class JavaBBinder : public BBinder
02 {
03 public:
04      ......
05
06      bool    checkSubclass(const void* subclassID) const
07      {
08          return subclassID == &gBinderOffsets;
09      }
10
11      ......
12 };
```

它检查参数subclassID是否指向全局变量gBinderOffsets。如果是，它的返回值就等于true；否则，它的返回值就等于false。

回到函数javaObjectForIBinder中，如果参数val指向的是一个JavaBBinder对象，那么调用它的成员函数checkSubclass得到的返回值就等于true。因此，这时候第7行就调用该JavaBBinder对象的成员函数object来获得一个Java对象，并且第9行将它返回给调用者。

JavaBBinder类的成员函数object的实现如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
01 class JavaBBinder : public BBinder
02 {
03 public:
04      ......
05
06      jobject object() const
07      {
08          return mObject;
09      }
```

```
10
11 private:
12     ......
13     jobject const   mObject;
14 };
```

在后面的5.10.3小节中，我们就会知道，JavaBBinder类的成员变量mObject指向的是Java层中的一个Binder对象，因此，JavaBBinder类的成员函数object返回给调用者的实际上是Java层中的一个Binder对象。

回到函数javaObjectForIBinder中，从前面的调用过程可以知道，这时候参数val指向的是一个句柄值等于0的Binder代理对象，因此，接下来就会执行第14行到第46行代码创建一个Java层的Java服务代理对象，即一个BinderProxy对象。

第17行调用Binder代理对象val的成员函数findObject来检查当前进程之前是否已经为它创建过一个BinderProxy对象了。如果是，第17行返回来的就是一个指向该BinderProxy对象的WeakReference对象object，即一个弱引用对象。由于弱引用对象object所指向的BinderProxy对象可能已经失效，因此，第19行就需要继续检查它的有效性，方法是调用它的成员函数get来获得一个强引用对象。如果能够成功地将弱引用对象object升级为一个强引用对象，即第20行的if语句为true，那么第22行就将该BinderProxy对象返回给调用者；否则，第25行就调用Binder代理对象val的成员函数detachObject来解除它与一个无效的BinderProxy对象的对应关系。

Binder代理对象内部有一个成员变量mObjects，它的类型为ObjectManager，用来管理与该Binder代理对象关联的外部对象。例如，函数javaObjectForIBinder在为一个Binder代理对象创建一个Java服务代理对象时，就会将创建出来的Java服务代理对象保存在该Binder代理对象的成员变量mObjects中，以便以后需要获取与该Binder代理对象对应的Java服务代理对象时，就可以通过它的成员变量mObjects来直接获得。由于一个Binder代理对象所关联的外部对象可能不止一个，因此，我们需要通过额外的一个参数来关联一个Binder代理对象和一个Java服务代理对象，这个参数就是全局变量gBinderProxyOffsets的地址值。

在继续分析函数javaObjectForIBinder的实现之前，我们首先介绍一个Binder代理对象是如何与一个Java服务代理对象关联起来的。

ObjectManager类的定义如下所示。

frameworks/base/include/binder/BpBinder.h

```
01 class BpBinder : public IBinder
02 {
03 public:
04     ......
05
06     class ObjectManager
07     {
08     public:
09                         ObjectManager();
10                         ~ObjectManager();
11
12         void            attach( const void* objectID,
13                                 void* object,
14                                 void* cleanupCookie,
15                                 IBinder::object_cleanup_func func);
16         void*           find(const void* objectID) const;
17         void            detach(const void* objectID);
18
19         void            kill();
20
21     private:
```

```
22                      ObjectManager(const ObjectManager&);
23          ObjectManager& operator=(const ObjectManager&);
24
25          struct entry_t
26          {
27              void* object;
28              void* cleanupCookie;
29              IBinder::object_cleanup_func func;
30          };
31
32          KeyedVector<const void*, entry_t> mObjects;
33      };
34      ......
35
36  private:
37      ......
38                       ObjectManager       mObjects;
39      ......
40  };
```

ObjectManager是定义在BpBinder内部的一个类，它有一个类型为KeyedVector的成员变量mObjects。这个成员变量以一个void指针作为关键字来维护与一个Binder代理对象所关联的外部对象，这些外部对象都使用一个entry_t结构体来描述。

结构体entry_t有三个成员变量：object、func和cleanupCookie，其中，object指向一个外部对象的一个弱引用对象；func是一个类型为IBinder::object_cleanup_func的函数指针，它是用来清理成员变量object所指向的外部对象的；cleanupCookie是函数指针func所指向的清理函数的一个调用参数。

当一个Binder代理对象被析构时，我们就需要清理与它所关联的外部对象，这是通过调用它的成员变量mObjects的成员函数kill来实现的，如下所示。

frameworks/base/libs/binder/BpBinder.cpp

```
01 void BpBinder::ObjectManager::kill()
02 {
03     const size_t N = mObjects.size();
04     ......
05     for (size_t i=0; i<N; i++) {
06         const entry_t& e = mObjects.valueAt(i);
07         if (e.func != NULL) {
08             e.func(mObjects.keyAt(i), e.object, e.cleanupCookie);
09         }
10     }
11
12     mObjects.clear();
13 }
```

第5行到第10行的for循环遍历ObjectManager类的成员变量mObjects所描述的一个外部对象列表。由于列表中的每一个entry_t结构体都描述了一个外部对象，因此，第8行就调用它的成员变量func所指向的一个函数来清理它们。

此外，ObjectManager类还提供了另外的三个成员函数：attach、find和detach，分别用来增加、查找和删除外部对象，它们的实现如下所示。

frameworks/base/libs/binder/BpBinder.cpp

```
01 void BpBinder::ObjectManager::attach(
02     const void* objectID, void* object, void* cleanupCookie,
03     IBinder::object_cleanup_func func)
04 {
05     entry_t e;
06     e.object = object;
07     e.cleanupCookie = cleanupCookie;
```

```
08      e.func = func;
09
10      if (mObjects.indexOfKey(objectID) >= 0) {
11          ......
12          return;
13      }
14
15      mObjects.add(objectID, e);
16  }
17
18  void* BpBinder::ObjectManager::find(const void* objectID) const
19  {
20      const ssize_t i = mObjects.indexOfKey(objectID);
21      if (i < 0) return NULL;
22      return mObjects.valueAt(i).object;
23  }
24
25  void BpBinder::ObjectManager::detach(const void* objectID)
26  {
27      mObjects.removeItem(objectID);
28  }
```

成员函数attach的参数object、cleanupCookie和func用来描述一个外部对象，即用来创建一个entry_t结构体；而参数objectID是用来标志前面三个参数所描述的一个外部对象的，因此，第15行就以它为关键字将前面创建的一个entry_t结构体保存在ObjectManager类的成员变量mObjects所描述的一个外部对象列表中。

成员函数find以参数objectID为关键字，在ObjectManager类的成员变量mObjects所描述的一个外部对象列表中找到一个对应的entry_t结构体，并且将这个entry_t结构体的成员变量object所指向的一个外部对象返回给调用者。

成员函数detach以参数objectID为关键字，在ObjectManager类的成员变量mObjects所描述的一个外部对象列表中删除一个对应的entry_t结构体，即解除一个外部对象与一个Binder代理对象的关联关系。

相应地，Binder代理对象也提供了三个成员函数：attachObject、findObject和detachObject，用来增加、查找和删除与它关联的外部对象，它们的实现如下所示。

frameworks/base/libs/binder/BpBinder.cpp

```
01  void BpBinder::attachObject(
02      const void* objectID, void* object, void* cleanupCookie,
03      object_cleanup_func func)
04  {
05      AutoMutex _l(mLock);
06      ......
07      mObjects.attach(objectID, object, cleanupCookie, func);
08  }
09
10  void* BpBinder::findObject(const void* objectID) const
11  {
12      AutoMutex _l(mLock);
13      return mObjects.find(objectID);
14  }
15
16  void BpBinder::detachObject(const void* objectID)
17  {
18      AutoMutex _l(mLock);
19      mObjects.detach(objectID);
20  }
```

它们分别是通过调用内部的成员变量mObjects的成员函数attach、find和detach来实现的，即调用前面所介绍的ObjectManager类的成员函数attach、find和detach来实现。

回到函数javaObjectForIBinder中，如果我们还没有为Binder代理对象val创建过Java服务代理对象，那么接下来第29行就会为它创建一个Java服务代理对象object。如果创建成功，即第30行的if语句为true，那么第33行就会将Binder代理对象val的地址值设置到前面所创建的Java服务代理对象object的成员变量mObject中。接着第34行增加Binder代理对象val的强引用计数，因为它被Java服务代理对象object引用了。

Java服务代理对象内部有一个成员变量mSelf，它指向了一个引用了该Java服务代理对象的弱引用对象，即一个WeakReference对象。第38行和第39行首先获得Java服务代理对象object内部的成员变量mSelf所指向的一个弱引用对象，接着再创建一个全局引用对象来引用它。接下来第40行和第41行就通过全局变量gBinderProxyOffsets将这个全局引用对象与Binder代理对象val关联起来，相当于将前面所创建的Java服务代理对象object与Binder代理对象val关联起来。这样，当我们以后再以Binder代理对象val为参数来调用函数javaObjectForIBinder时，就可以直接获得保存在它内部的成员变量mObjects中的Java服务代理对象了。

最后，第46行就可以将前面所创建的Java服务代理对象object返回给调用者了。

函数javaObjectForIBinder执行完成之后，先返回到BinderInternal类的静态成员函数getContextObject中，接着再返回到ServiceManager类的静态成员函数getIServiceManager中，这时候我们就获得一个句柄值等于0的Java服务代理对象，接下来就可以调用ServiceManagerNative类的静态成员函数asInterface将它封装成一个Service Manager的Java代理对象了。

ServiceManagerNative类的静态成员函数asInterface的实现如下所示。

frameworks/base/core/java/android/os/ServiceManagerNative.java

```
01 public abstract class ServiceManagerNative extends Binder implements IServiceManager
02 {
03      ......
04
05      static public IServiceManager asInterface(IBinder obj)
06      {
07          if (obj == null) {
08              return null;
09          }
10          IServiceManager in =
11              (IServiceManager)obj.queryLocalInterface(descriptor);
12          if (in != null) {
13              return in;
14          }
15
16          return new ServiceManagerProxy(obj);
17      }
18
19      ......
20 }
```

参数obj指向一个Java服务代理对象，即一个BinderProxy对象，它的成员函数queryLocalInterface的返回值为null，因此，第16行就会先创建一个ServiceManagerProxy对象，即一个Service Manager的Java代理对象，接着将它返回给调用者。

回到ServiceManager类的静态成员函数getIServiceManager中，接着它就可以将获得的Service Manager的Java代理对象保存在其静态成员变量sServiceManager中，最后将它的IServiceManager接口返回给调用者。

5.10.2 Java服务接口的定义和解析

在实现自己的Java服务之前，首先要定义这个Java服务要实现的接口，即定义自己的Java服

务接口。在Android应用程序中，我们可以通过Android接口描述语言（Android Interface Definition Language，AIDL）来定义Java服务接口。AIDL是一种Binder进程间通信接口的描述语言，通过它来定义的Java服务接口具有执行Binder进程间通信的能力。

以AIDL定义的Java服务接口保存在一个以“.aidl”为后缀名的文件中，在编译时，它们会被转化为使用Java语言来描述的Java服务接口。学习这些使用Java语言描述的Java服务接口有助于我们进一步理解Binder进程间通信机制的Java接口，因此，接下来我们就以2.4.1小节中介绍的硬件访问服务接口IFregService为例，详细分析这些使用AIDL定义的Java服务接口是如何被编译为使用Java语言来描述的Java服务接口的。

硬件访问服务接口IFregService定义在文件IFregService.aidl中，文件内容如下所示。

frameworks/base/core/java/android/os/IFregService.aidl

```
1 package android.os;
2
3 interface IFregService {
4         void setVal(int val);
5         int getVal();
6 }
```

硬件访问服务接口IFregService有两个成员函数：setVal和getVal，分别用来写和读虚拟硬件设备freg的寄存器val。

在编译文件IFregService.aidl时，编译系统会生成一个中间文件IFregService.java，它的内容如下所示。

out/target/common/obj/JAVA_LIBRARIES/framework_intermediates/src/core/java/android/os/IFregService.java

```
001 /*
002  * This file is auto-generated.  DO NOT MODIFY.
003  * Original file: frameworks/base/core/java/android/os/IFregService.aidl
004  */
005 package android.os;
006
007 public interface IFregService extends android.os.IInterface {
008     /** Local-side IPC implementation stub class. */
009     public static abstract class Stub extends android.os.Binder implements android.os.IFregService {
010          private static final java.lang.String DESCRIPTOR = "android.os.IFregService";
011          /** Construct the stub at attach it to the interface. */
012          public Stub() {
013                 this.attachInterface(this, DESCRIPTOR);
014          }
015
016          /**
017          * Cast an IBinder object into an android.os.IFregService interface,
018          * generating a proxy if needed.
019          */
020          public static android.os.IFregService asInterface(android.os.IBinder obj) {
021                 if ((obj==null)) {
022                        return null;
023                 }
024
025                 android.os.IInterface iin = (android.os.IInterface)obj.queryLocalInterface(DESCRIPTOR);
026                 if (((iin!=null)&&(iin instanceof android.os.IFregService))) {
027                        return ((android.os.IFregService)iin);
028                 }
029
030                 return new android.os.IFregService.Stub.Proxy(obj);
031          }
032
033          public android.os.IBinder asBinder() {
034                 return this;
035          }
```

```
036
037         @Override
038         public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel
reply, int flags) throws android.os.RemoteException {
039                 switch (code) {
040                 case INTERFACE_TRANSACTION: {
041                         reply.writeString(DESCRIPTOR);
042                         return true;
043                 }
044                 case TRANSACTION_setVal: {
045                         data.enforceInterface(DESCRIPTOR);
046                         int _arg0;
047                         _arg0 = data.readInt();
048                         this.setVal(_arg0);
049                         reply.writeNoException();
050                         return true;
051                 }
052                 case TRANSACTION_getVal: {
053                         data.enforceInterface(DESCRIPTOR);
054                         int _result = this.getVal();
055                         reply.writeNoException();
056                         reply.writeInt(_result);
057                         return true;
058                 }
059                 }
060
061                 return super.onTransact(code, data, reply, flags);
062         }
063
064         private static class Proxy implements android.os.IFregService {
065                 private android.os.IBinder mRemote;
066
067                 Proxy(android.os.IBinder remote) {
068                         mRemote = remote;
069                 }
070
071                 public android.os.IBinder asBinder() {
072                         return mRemote;
073                 }
074
075                 public java.lang.String getInterfaceDescriptor() {
076                         return DESCRIPTOR;
077                 }
078
079                 public void setVal(int val) throws android.os.RemoteException {
080                         android.os.Parcel _data = android.os.Parcel.obtain();
081                         android.os.Parcel _reply = android.os.Parcel.obtain();
082
083                         try {
084                                 _data.writeInterfaceToken(DESCRIPTOR);
085                                 _data.writeInt(val);
086                                 mRemote.transact(Stub.TRANSACTION_setVal, _data, _reply, 0);
087                                 _reply.readException();
088                         } finally {
089                                 _reply.recycle();
090                                 _data.recycle();
091                         }
092                 }
093
094                 public int getVal() throws android.os.RemoteException {
095                         android.os.Parcel _data = android.os.Parcel.obtain();
096                         android.os.Parcel _reply = android.os.Parcel.obtain();
097                         int _result;
```

```
098
099                             try {
100                                     _data.writeInterfaceToken(DESCRIPTOR);
101                                     mRemote.transact(Stub.TRANSACTION_getVal, _data, _reply,
0);
102                                     _reply.readException();
103                                     _result = _reply.readInt();
104                             } finally {
105                                     _reply.recycle();
106                                     _data.recycle();
107                             }
108
109                             return _result;
110                     }
111             }
112
113             static final int TRANSACTION_setVal = (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);
114             static final int TRANSACTION_getVal = (android.os.IBinder.FIRST_CALL_TRANSACTION + 1);
115     }
116
117     public void setVal(int val) throws android.os.RemoteException;
118     public int getVal() throws android.os.RemoteException;
119 }
```

文件定义了一个Java接口IFregService、一个Java抽象类IFregService.Stub和一个Java类IFregService.Stub.Proxy，其中，Java抽象类IFregService.Stub是在Java接口IFregService内部定义的，而Java类IFregService.Stub.Proxy是在Java抽象类IFregService.Stub内部定义的。

IFregService.Stub和IFregService.Stub.Proxy类都实现了IFregService接口，其中，IFregService.Stub类用来描述一个Java服务，而IFregService.Stub.Proxy类用来描述一个Java服务代理对象。我们要实现的硬件访问服务FregService就是以IFregService.Stub为父类的，并且实现了它的IFregService接口的两个成员函数setVal和getVal。

IFregService.Stub类的静态成员函数asInterface通常用来将一个Java服务代理对象，即一个BinderProxy对象封装成一个IFregService.Stub.Proxy对象，即一个实现了IFregService接口的Java服务代理对象。

IFregService.Stub类的成员函数onTransact是用来接收和分发进程间通信请求的。当Client进程向一个实现了IFregService接口的Java服务发送一个类型为TRANSACTION_setVal或者TRANSACTION_getVal的进程间通信请求时，IFregService.Stub类的成员函数onTransact就会被调用，它分别将这两种类型的进程间通信请求分发给由其子类所重写的两个成员函数setVal和getVal来处理。

IFregService.Stub.Proxy类内部有一个成员变量mRemote，它指向的是一个Java服务代理对象，即一个BinderProxy对象，用来向一个实现了IFregService接口的Java服务发送进程间通信请求。

IFregService.Stub.Proxy类实现了IFregService接口的两个成员函数：setVal和getVal，它们分别向一个实现了IFregService接口的Java服务发送一个类型为TRANSACTION_setVal和TRANSACTION_getVal的进程间通信请求。

得到了使用Java语言描述的Java服务接口之后，接下来我们继续分析Java服务的启动过程、Java服务代理对象的获取过程，以及Java服务的调用过程。

5.10.3　Java服务的启动过程

Java服务一般是运行在Android系统进程System或者Android应用程序进程中的，它们在启动之前，同样需要将自己注册到Service Manager中，以便Client进程可以通过Service Manager来获得它们的Java服

务代理对象。

由于Android系统进程System和Android应用程序进程在启动时会在内部启动一个Binder线程池[1]，因此，运行在它们里面的Java服务在启动时，只需要将自己注册到Service Manager中就可以了。接下来，我们就以2.4小节介绍的硬件访问服务FregService为例，详细分析一个Java服务的启动过程。

硬件访问服务FregService是从IFregService.Stub类继承下来的，它是一个实现了IFregService接口的Java服务，它的定义如下所示。

frameworks/base/services/java/com/android/server/FregService.java

```
1 package com.android.server;
2
3 import android.content.Context;
4 import android.os.IFregService;
5 import android.util.Slog;
6
7 public class FregService extends IFregService.Stub {
8     ......
9 };
```

硬件访问服务FregService是在Android系统进程System中启动的，如下所示。

frameworks/base/services/java/com/android/server/SystemServer.java

```
01 class ServerThread extends Thread {
02     ......
03
04     @Override
05     public void run() {
06         ......
07
08         try {
09             Slog.i(TAG, "Freg Service");
10             ServiceManager.addService("freg", new FregService());
11         } catch (Throwable e) {
12             Slog.e(TAG, "Failure starting Freg Service", e);
13         }
14
15         ......
16     }
17 }
```

Android系统进程System在启动时，会创建一个ServerThread 线程来启动系统中的关键服务，硬件访问服务FregService就是在这个ServerThread 线程中启动的。第10行首先创建一个硬件访问服务FregService，接着再调用ServiceManager类的静态成员函数addService将它注册到Service Manager中。

接下来，我们首先分析硬件访问服务FregService的创建过程，然后再分析它的注册过程。

由于硬件访问服务FregService继承了IFregService.Stub类，而后者又继承了Binder类，因此，硬件访问服务FregService在创建的过程中，会调用Binder类的构造函数来执行初始化操作。

Binder类的构造函数的实现如下所示。

frameworks/base/core/java/android/os/Binder.java

```
01 public class Binder implements IBinder {
02     ......
03
04     private int mObject;
05
06     ......
07
08     public Binder() {
```

1 在后面的第11章和第12章中，我们将详细分析Android系统进程System和Android应用程序进程的启动过程。

```
09          init();
10          ......
11      }
12
13      private native final void init();
14
15      ......
16  }
```

它主要是调用了成员函数init来执行初始化工作。Binder类的成员函数init是一个JNI方法，它的实现如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
1 static void android_os_Binder_init(JNIEnv* env, jobject clazz)
2 {
3     JavaBBinderHolder* jbh = new JavaBBinderHolder(env, clazz);
4     ......
5
6     jbh->incStrong(clazz);
7     env->SetIntField(clazz, gBinderOffsets.mObject, (int)jbh);
8 }
```

参数clazz指向的是前面在Java层中创建的硬件访问服务FregService，第3行以它为参数在C++层中创建了一个JavaBBinderHolder对象jbh，接着第6行代码增加了JavaBBinderHolder对象jbh的强引用计数，因为它被Java层中的硬件访问服务FregService引用了。最后，第7行代码将JavaBBinderHolder对象jbh的地址值保存在硬件访问服务FregService的父类Binder的成员变量mObject中。这样，运行在Java层中的硬件访问服务FregService就可以通过它的成员变量mObject来访问运行在C++层中的JavaBBinderHolder对象jbh了。

JavaBBinderHolder对象jbh的创建过程如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
01 class JavaBBinderHolder : public RefBase
02 {
03 public:
04     JavaBBinderHolder(JNIEnv* env, jobject object)
05         : mObject(object)
06     {
07         ......
08     }
09     ......
10
11 private:
12     ......
13     jobject         mObject;
14     wp<JavaBBinder> mBinder;
15 };
```

它将参数object保存在自己的成员变量mObject中。从前面的调用过程可以知道，参数object指向的是一个在Java层中创建的硬件访问服务FregService。这样，运行在C++层中的JavaBBinderHolder对象jbh就可以通过它的成员变量mObject来访问运行在Java层中的硬件访问服务FregService了。

JavaBBinderHolder类还有一个成员变量mBinder，它指向的是一个JavaBBinder对象。JavaBBinder类继承了Binder库中的BBinder类，因此，它可以用来描述一个Binder本地对象。

至此，硬件访问服务FregService的创建过程就结束了。从它的创建过程可以知道，它在C++层中有一个对应的JavaBBinderHolder对象，而这个JavaBBinderHolder对象通过其成员变量mBinder持有一个类型为JavaBBinder的Binder本地对象。接下来我们就会看到，硬件访问服务FregService的进程间通信功能就是通过与其对应的JavaBBinderHolder对象所持有的一个类型为JavaBBinder的Binder本地对象来

实现的。

硬件访问服务FregService创建完成之后，返回到ServerThread类的成员函数run中，接下来就调用ServiceManager类的静态成员函数addService将它注册到Service Manager中。

ServiceManager类的静态成员函数addService的实现如下所示。

frameworks/base/core/java/android/os/ServiceManager.java

```
01 public final class ServiceManager {
02     ......
03
04     public static void addService(String name, IBinder service) {
05         try {
06             getIServiceManager().addService(name, service);
07         } catch (RemoteException e) {
08             Log.e(TAG, "error in addService", e);
09         }
10     }
11
12     ......
13 }
```

第6行首先调用ServiceManager类的静态成员函数getIServiceManager来获得一个Service Manager的Java代理对象，接着再调用这个Java代理对象的成员函数addService将Java服务service注册到Service Manager中，并且指定这个Java服务service的注册名称为name。

Service Manager的Java代理对象的成员函数addService的实现如下所示。

frameworks/base/core/java/android/os/ServiceManagerNative.java

```
01 class ServiceManagerProxy implements IServiceManager {
02     ......
03
04     public void addService(String name, IBinder service)
05         throws RemoteException {
06             Parcel data = Parcel.obtain();
07             Parcel reply = Parcel.obtain();
08             data.writeInterfaceToken(IServiceManager.descriptor);
09             data.writeString(name);
10             data.writeStrongBinder(service);
11             mRemote.transact(ADD_SERVICE_TRANSACTION, data, reply, 0);
12             reply.recycle();
13             data.recycle();
14     }
15
16     ......
17     private IBinder mRemote;
18 }
```

类似于C++层中的Parcel类，Java层中的Parcel类也是用来封装进程间通信数据的。Java层中的每一个Parcel对象在C++层中都有一个对应的Parcel对象，后者的地址值就保存在前者的成员变量mObject中。当我们将进程间通信数据封装在一个Java层中的Parcel对象时，这个Java层中的Parcel对象就会通过其成员变量mObject找到与它对应的运行在C++层中的Parcel对象，并且将这些进程间通信数据封装到C++层中的Parcel对象里面去。

第6行和第7行创建了两个Parcel对象data和reply，其中，Parcel对象data是用来封装进程间通信数据的，而Parcel对象reply是用来封装进程间通信结果数据的。第8行首先调用Parcel对象data的成员函数writeInterfaceToken写入一个进程间通信头，接着第9行调用Parcel对象data的成员函数writeString写入一个即将要注册的Java服务的名称，最后第10行调用Parcel对象data的成员函数wrtieStrongBinder写入一个即将要注册的Java服务。下面我们就着重分析即将要注册的Java服务，即硬件访问服务FregService是如何写入到Parcel对象data中的。

Java层中的Parcel类的成员函数wrtieStrongBinder是一个JNI方法，它的定义如下所示。

frameworks/base/core/java/android/os/Parcel.java

```
1 public final class Parcel {
2     ......
3
4     public final native void writeStrongBinder(IBinder val);
5
6     ......
7 }
```

它对应的实现函数为android_os_Parcel_writeStrongBinder，如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
1 static void android_os_Parcel_writeStrongBinder(JNIEnv* env, jobject clazz, jobject object)
2 {
3     Parcel* parcel = parcelForJavaObject(env, clazz);
4     if (parcel != NULL) {
5         const status_t err = parcel->writeStrongBinder(ibinderForJavaObject(env, object));
6         ......
7     }
8 }
```

参数clazz指向的是一个Java层中的Parcel对象，第3行调用函数parcelForJavaObject获得一个与它对应的、运行在C++层中的Parcel对象parcel。

参数object指向的是即将要注册的Java服务，第5行首先调用函数ibinderForJavaObject来获得一个与它对应的Binder本地对象，然后再将这个Binder本地对象写入到前面得到的Parcel对象parcel中。在前面的5.8.1.1小节中，我们已经分析过C++层中的Parcel类的成员函数writeStrongBinder的实现了，因此，接下来我们就重点分析函数ibinderForJavaObject是如何根据参数object来获得一个对应的Binder本地对象的。

函数ibinderForJavaObject的实现如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
01 sp<IBinder> ibinderForJavaObject(JNIEnv* env, jobject obj)
02 {
03     if (obj == NULL) return NULL;
04
05     if (env->IsInstanceOf(obj, gBinderOffsets.mClass)) {
06         JavaBBinderHolder* jbh = (JavaBBinderHolder*)
07             env->GetIntField(obj, gBinderOffsets.mObject);
08         return jbh != NULL ? jbh->get(env) : NULL;
09     }
10
11     if (env->IsInstanceOf(obj, gBinderProxyOffsets.mClass)) {
12         return (IBinder*)
13             env->GetIntField(obj, gBinderProxyOffsets.mObject);
14     }
15
16     ......
17     return NULL;
18 }
```

参数obj要么指向一个Java服务，要么指向一个Java服务代理对象；否则，第17行就给调用者返回一个NULL值，表示它不能将参数obj转换成一个Binder本地对象或者一个Binder代理对象。

如果参数obj指向的是一个Java服务代理对象，即第11行的if语句为true，那么第12行和第13行代码就找到与它对应的一个Binder代理对象，并且将这个Binder代理对象返回给调用者。在前面的5.10.1小节中提到，Java层中的每一个Java服务代理对象在C++层中都有一个对应的Binder代理对象，并且这个Binder代理对象的地址值就保存在它的成员变量mObject中。因此，第12行和第13行代码可以轻易地

找到与参数obj对应的Binder代理对象。

从前面的调用过程可以知道，参数obj指向的是即将要注册的硬件访问服务FregService，它是一个Java服务，因此，第5行的if语句为true，函数接下来会执行第6行到第8行代码来获得一个Binder本地对象。

前面提到，一个Java服务的成员变量mObject中保存的是与其对应的一个JavaBBinderHolder对象，因此，第6行和第7行代码就可以安全地将参数obj的成员变量mObject转换为一个JavaBBinderHolder对象jbh。接着第8行调用JavaBBinderHolder对象jbh的成员函数get来获得一个Binder本地对象。

JavaBBinderHolder类的成员函数get的实现如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
01 class JavaBBinderHolder : public RefBase
02 {
03     ......
04
05     sp<JavaBBinder> get(JNIEnv* env)
06     {
07         AutoMutex _l(mLock);
08         sp<JavaBBinder> b = mBinder.promote();
09         if (b == NULL) {
10             b = new JavaBBinder(env, mObject);
11             mBinder = b;
12             ......
13         }
14
15         return b;
16     }
17
18     ......
19 };
```

前面提到，JavaBBinderHolder类的成员变量mBinder指向的是一个类型为JavaBBinder的Binder本地对象，而成员变量mObject指向的是一个Java服务。

JavaBBinderHolder类的成员函数get用来将成员变量mBinder所指向的一个JavaBBinder对象返回给调用者。由于成员变量mBinder是一个类型为JavaBBinder的弱指针，即此时它指向的JavaBBinder对象可能已经被销毁，因此，在将它返回给调用者之前，首先要将它升级为一个类型为JavaBBinder的强指针。如果升级成功，那么第15行就可以直接将成员变量mBinder所指向的一个JavaBBinder对象返回给调用者了；否则，第10行和第11行就需要先创建一个JavaBBinder对象保存在成员变量mBinder中，接着第15行再将这个JavaBBinder对象返回给调用者。

类型为JavaBBinder的Binder本地对象内部有一个成员变量mObject，它指向了一个Java服务，它是在JavaBBinder类的构造函数中被始化的，如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
01 class JavaBBinder : public BBinder
02 {
03 public:
04     JavaBBinder(JNIEnv* env, jobject object)
05         : mVM(jnienv_to_javavm(env)), mObject(env->NewGlobalRef(object))
06     {
07         ......
08     }
09     ......
10
11 private:
12     JavaVM* const   mVM;
13     jobject const   mObject;
14 };
```

从前面的调用过程可以知道，一个JavaBBinder对象的成员变量mObject所指向的Java服务实际上就是持有该JavaBBinder对象的JavaBBinderHolder对象的成员变量mObject所指向的Java服务，因此，这时候我们就将一个Java服务分别与一个JavaBBinderHolder对象和一个JavaBBinder对象关联起来。

返回到Service Manager的Java代理对象的成员函数addService中，将要传递给Service Manager的进程间通信数据写入到Parcel对象data中之后，接着就调用它的成员变量mRemote的成员函数transact向Service Manager发送一个类型为ADD_SERVICE_TRANSACTION的进程间通信请求，即请求Service Manager将一个Java服务注册到它里面去。从前面5.10.1小节的内容可以知道，Service Manager的Java代理对象的成员变量mRemote指向的是一个句柄值等于0的Java服务代理对象，因此，接下来就会执行BinderProxy类的成员函数transact来向Service Manager发送一个进程间通信请求。

BinderProxy类的成员函数transact是一个JNI方法，它的定义如下所示。

frameworks/base/core/java/android/os/Binder.java

```
1 final class BinderProxy implements IBinder {
2     ......
3
4     public native boolean transact(int code, Parcel data, Parcel reply,
5                                    int flags) throws RemoteException;
6
7     ......
8 }
```

它对应的实现函数为android_os_BinderProxy_transact，如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
01 static jboolean android_os_BinderProxy_transact(JNIEnv* env, jobject obj,
02                         jint code, jobject dataObj,
03                         jobject replyObj, jint flags)
04 {
05     ......
06
07     Parcel* data = parcelForJavaObject(env, dataObj);
08     ......
09     Parcel* reply = parcelForJavaObject(env, replyObj);
10     ......
11
12     IBinder* target = (IBinder*)
13         env->GetIntField(obj, gBinderProxyOffsets.mObject);
14     ......
15
16     status_t err = target->transact(code, *data, reply, flags);
17     ......
18
19     if (err == NO_ERROR) {
20         return JNI_TRUE;
21     } else if (err == UNKNOWN_TRANSACTION) {
22         return JNI_FALSE;
23     }
24
25     signalExceptionForError(env, obj, err);
26     return JNI_FALSE;
27 }
```

第7行和第9行分别将Java层中的两个Parcel对象dataObj和replyObj转换为C++层中的Parcel对象data和reply，前者包含了要传递给Service Manager的进程间通信数据，而后者用来保存从Service Manager返回来的进程间通信结果数据。

参数obj指向的是一个Java服务代理对象，第12行和第13行将它的成员变量mObject转换为一个Binder代理对象target。从前面的调用过程可以知道，前面得到的Binder代理对象target是一个句柄值等于0的Binder代理对象。

第16行调用Binder代理对象target的成员函数transact向Server Manager发送一个类型为code的进程间通信请求，即一个类型为ADD_SERVICE_TRANSACTION的进程间通信请求。Service Manager接收到这个进程间通信请求之后，就会将保存在Parcel对象data中的一个Binder本地对象的信息取出来，并且注册到它内部去。将一个Binder本地对象注册到Service Manager中的过程可以参考前面5.8.1小节的内容，这里就不详述了。

至此，Java服务的启动过程就分析完成了。从这个过程可以看出，在注册一个Java服务时，我们并不是真的将这个Java服务注册到Service Manager中，而是将与它对应的一个类型为JavaBBinder的Binder本地对象注册到Service Manager中。这样，当这个类型为JavaBBinder的Binder本地对象接收到来自Client进程的进程间通信请求时，它就需要将这些进程间通信请求交给它所对应的Java服务来处理。

5.10.4 Java服务代理对象的获取过程

Java服务将自己注册到Service Manager中之后，Android应用程序就可以通过Service Manager来获得它的一个Java服务代理对象了。有了这个Java服务代理对象之后，Android应用程序就可以使用相应的Java服务了。在本节中，我们将以2.5小节中介绍的Android应用程序Freg为例，详细分析一个Java服务代理对象的获取过程。

Android应用程序Freg获取硬件访问服务FregService的一个Java代理对象的代码片段如下所示。

packages/experimental/Freg/src/shy/luo/freg/Freg.java

```
01 public class Freg extends Activity implements OnClickListener {
02         ......
03
04         private IFregService fregService = null;
05         ......
06
07         @Override
08         public void onCreate(Bundle savedInstanceState) {
09                 ......
10
11                 fregService = IFregService.Stub.asInterface(
12                         ServiceManager.getService("freg"));
13
14                 ......
15         }
16
17         ......
18 }
```

第11行和第12行代码首先通过ServiceManager类的静态成员函数getService来获得一个名称为“freg”的Java服务代理对象，接着再调用IFregService.Stub类的静态成员函数asInterface将这个Java服务代理对象封装成一个实现了IFregService接口的Java服务代理对象。由于前面我们注册的硬件访问服务FregService的名称为“freg”，因此，第12行代码调用ServiceManager类的静态成员函数getService就可以获得硬件访问服务FregService的一个Java代理对象了。

接下来，我们就分别分析ServiceManager类的静态成员函数getService和IFregService.Stub类的静态成员函数asInterface的实现。

ServiceManager类的静态成员函数getService的实现如下所示。

frameworks/base/core/java/android/os/ServiceManager.java

```
01 public final class ServiceManager {
02     ......
03
```

```
04      public static IBinder getService(String name) {
05          try {
06              IBinder service = sCache.get(name);
07              if (service != null) {
08                  return service;
09              } else {
10                  return getIServiceManager().getService(name);
11              }
12          } catch (RemoteException e) {
13              Log.e(TAG, "error in getService", e);
14          }
15          return null;
16      }
17
18      ......
19  }
```

ServiceManager类的静态成员变量sCache用来保存前面已经获得的Java服务代理对象，这些Java服务代理对象是以它们所引用的Java服务的名称作为关键字来保存的，因此，第6行首先在ServiceManager类的静态成员变量sCache中检查是否已经存在一个与参数name对应的Java服务代理对象。如果存在，即第7行的if语句为true，那么第8行就直接将前面获得的Java服务代理对象service返回给调用者；否则，第10行就首先调用ServiceManager类的静态成员函数getIServiceManager来获得一个Service Manager的Java代理对象，接着再调用这个Service Manager的Java代理对象的成员函数getService来获得与参数name对应的Java服务代理对象。

Service Manager的Java代理对象的成员函数getService的实现如下所示。

frameworks/base/core/java/android/os/ServiceManagerNative.java

```
01  class ServiceManagerProxy implements IServiceManager {
02      ......
03
04      public IBinder getService(String name) throws RemoteException {
05          Parcel data = Parcel.obtain();
06          Parcel reply = Parcel.obtain();
07          data.writeInterfaceToken(IServiceManager.descriptor);
08          data.writeString(name);
09          mRemote.transact(GET_SERVICE_TRANSACTION, data, reply, 0);
10          IBinder binder = reply.readStrongBinder();
11          reply.recycle();
12          data.recycle();
13          return binder;
14      }
15
16      ......
17      private IBinder mRemote;
18  }
```

与在前面5.10.3小节中分析的Service Manager的Java代理对象的成员函数addService的实现类似，Service Manager的Java代理对象的成员函数getService也是通过内部的一个句柄值为0的Java代理对象来与Service Manager通信的。两者的区别在于请求Service Manager执行的操作不同，其中，前者请求Service Manager执行一个ADD_SERVICE_TRANSACTION操作，而后者请求Service Manager执行一个GET_SERVICE_TRANSACTION操作。

Service Manager处理完成GET_SERVICE_TRANSACTION操作之后，就会将一个Binder代理对象保存在Parcel对象reply中。因此，第10行就调用Parcel对象reply的成员函数readStrongBinder将它读取出来，以及将它转换为一个Java服务代理对象。

Java层中的Parcel类的成员函数readStrongBinder是一个JNI方法，它的定义如下所示。

frameworks/base/core/java/android/os/Parcel.java

```
1 public final class Parcel {
2     ......
3
4     public final native IBinder readStrongBinder();
5
6     ......
7 }
```

它对应的实现函数为android_os_Parcel_readStrongBinder，如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
1 static jobject android_os_Parcel_readStrongBinder(JNIEnv* env, jobject clazz)
2 {
3     Parcel* parcel = parcelForJavaObject(env, clazz);
4     if (parcel != NULL) {
5         return javaObjectForIBinder(env, parcel->readStrongBinder());
6     }
7     return NULL;
8 }
```

参数clazz指向的是一个Java层中的Parcel对象，因此，第3行调用函数parcelForJavaObject获得一个与它对应的、运行在C++层中的Parcel对象parcel。

由于Parcel对象parcel里面保存了一个Binder代理对象，因此，第5行首先调用它的成员函数readStrongBinder将这个Binder代理对象读取出来，接着再调用函数javaObjectForIBinder获得一个对应的Java服务代理对象，最后将这个Java服务代理对象返回给调用者。C++层中Parcel类的成员函数readStrongBinder以及函数javaObjectForIBinder的实现可以参考前面的5.9和5.10.1小节的内容，这里不再详述。

获得了一个Java服务代理对象之后，返回到Activity组件Freg的成员函数onCreate中，接着它就调用IFregService.Stub类的静态成员函数asInterface将这个Java服务代理对象封装成一个实现了IFregService接口的Java服务代理对象，即一个IFregService.Stub.Proxy对象。

IFregService.Stub类的静态成员函数asInterface的实现如下所示。

out/target/common/obj/JAVA_LIBRARIES/framework_intermediates/src/core/java/android/os/IFregService.java

```
01 public interface IFregService extends android.os.IInterface {
02     /** Local-side IPC implementation stub class. */
03     public static abstract class Stub extends android.os.Binder implements android.os.IFregService {
04             ......
05
06             public static android.os.IFregService asInterface(android.os.IBinder obj) {
07                     if ((obj==null)) {
08                             return null;
09                     }
10
11                     android.os.IInterface iin = (android.os.IInterface)obj.queryLocalInterface(DESCRIPTOR);
12                     if (((iin!=null)&&(iin instanceof android.os.IFregService))) {
13                             return ((android.os.IFregService)iin);
14                     }
15
16                     return new android.os.IFregService.Stub.Proxy(obj);
17             }
18
19             ......
20     }
21
22     ......
23 }
```

从前面的调用过程可以知道，参数obj指向的是一个Java服务代理对象，即一个BinderProxy对象。由于BinderProxy类的成员函数queryLocalInterface的返回值为null，因此，第16行就会将参数obj所指向的一个Java服务代理对象封装成一个IFregService.Stub.Proxy对象，最后再将这个IFregService.Stub.Proxy对象返回给调用者。

至此，一个Java服务代理对象的获取过程就分析完成了。在接下来的5.10.5小节中，我们继续分析Java服务的调用过程。

5.10.5　Java服务的调用过程

Android应用程序获得了一个Java服务代理对象之后，就可以通过Binder进程间通信机制使用这个Java服务了。在本节中，我们将以2.5小节介绍的Android应用程序Freg为例，详细分析Java服务的调用过程。

Android应用程序Freg获取了硬件访问服务FregService的一个Java代理对象之后，就可以调用这个实现了IFregService接口的Java代理对象的成员函数getVal和setVal来读写虚拟硬件设备freg的寄存器val了，如下面的第12行和第20行代码所示。

packages/experimental/Freg/src/shy/luo/freg/Freg.java

```
01 public class Freg extends Activity implements OnClickListener {
02         ......
03
04         private IFregService fregService = null;
05
06         ......
07
08         @Override
09         public void onClick(View v) {
10                 if(v.equals(readButton)) {
11                         try {
12                                 int val = fregService.getVal();
13                                 ......
14                         } catch (RemoteException e) {
15                                 ......
16                         }
17                 } else if(v.equals(writeButton)) {
18                         try {
19                                 ......
20                                 fregService.setVal(val);
21                         } catch (RemoteException e) {
22                                 ......
23                         }
24                 }
25                 ......
26           }
27 }
```

接下来，我们就详细分析第12行调用硬件访问服务FregService的Java代理对象fregService的成员函数getVal来读取虚拟硬件设备freg的寄存器val的过程。

从前面5.10.4小节的内容可以知道，硬件访问服务FregService的Java代理对象的类型为IFregService.Stub.Proxy，它的成员函数getVal的实现如下所示。

out/target/common/obj/JAVA_LIBRARIES/framework_intermediates/src/core/java/android/os/IFregService.java

```
01 public interface IFregService extends android.os.IInterface {
02           /** Local-side IPC implementation stub class. */
03           public static abstract class Stub extends android.os.Binder implements android.os.IFregService {
04                ......
```

```
05                private static class Proxy implements android.os.IFregService {
06                        private android.os.IBinder mRemote;
07                        ......
08                        public int getVal() throws android.os.RemoteException {
09                                android.os.Parcel _data = android.os.Parcel.obtain();
10                                android.os.Parcel _reply = android.os.Parcel.obtain();
11                                int _result;
12
13                                try {
14                                        _data.writeInterfaceToken(DESCRIPTOR);
15                                        mRemote.transact(Stub.TRANSACTION_getVal, _data, _reply, 0);
16                                        _reply.readException();
17                                        _result = _reply.readInt();
18                                } finally {
19                                        _reply.recycle();
20                                        _data.recycle();
21                                }
22
23                                return _result;
24                        }
25                 }
26          }
27
28          ......
29 }
```

硬件访问服务FregService的Java代理对象的成员函数getVal首先将进程间通信数据写入到一个Parcel对象_data中，接着再调用内部的一个Java代理对象mRemote的成员函数transact来向硬件访问服务FregService发送一个类型为Stub.TRANSACTION_getVal的进程间通信请求。硬件访问服务FregService处理完成这个进程间通信请求之后，就将进程间通信结果数据返回来，保存在另外一个Parcel对象_reply中。

在前面的5.10.3小节中，由于我们是将一个与硬件访问服务FregService相关联的类型为JavaBBinder的Binder本地对象注册到Service Manager中，因此，Binder驱动程序就会将应用程序Freg发送过来的Stub.TRANSACTION_getVal进程间通信请求分发给这个类型为JavaBBinder的Binder本地对象的成员函数onTransact来处理。

JavaBBinder类的成员函数onTransact的实现如下所示。

frameworks/base/core/jni/android_util_Binder.cpp

```
01 class JavaBBinder : public BBinder
02 {
03     ......
04
05     virtual status_t onTransact(
06         uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags = 0)
07     {
08         JNIEnv* env = javavm_to_jnienv(mVM);
09
10         ......
11
12         jboolean res = env->CallBooleanMethod(mObject, gBinderOffsets.mExecTransact,
13             code, (int32_t)&data, (int32_t)reply, flags);
14
15         ......
16
17         return res != JNI_FALSE ? NO_ERROR : UNKNOWN_TRANSACTION;
```

```
18      }
19
20      ......
21
22      JavaVM* const   mVM;
23      jobject const   mObject;
24  };
```

在前面的5.10.3小节中提到，JavaBBinder类的成员变量mObject指向的是一个Java服务。在这个情景中，这个Java服务便是硬件访问服务FregService，因此，第12行代码就将Binder驱动程序中发送过来的进程间通信请求分发给硬件访问服务FregService的成员函数execTransact来处理。硬件访问服务FregService继承了Binder类，并且继承了它的成员函数execTransact，因此，第12行代码实际上是调用了Binder类的成员函数execTransact。

Binder类的成员函数execTransact的实现如下所示。

frameworks/base/core/java/android/os/Binder.java

```
01 public class Binder implements IBinder {
02      ......
03
04      // Entry point from android_util_Binder.cpp's onTransact
05      private boolean execTransact(int code, int dataObj, int replyObj, int flags) {
06          Parcel data = Parcel.obtain(dataObj);
07          Parcel reply = Parcel.obtain(replyObj);
08          // theoretically, we should call transact, which will call onTransact,
09          // but all that does is rewind it, and we just got these from an IPC,
10          // so we'll just call it directly.
11          boolean res;
12          try {
13              res = onTransact(code, data, reply, flags);
14          } catch (RemoteException e) {
15              reply.writeException(e);
16              res = true;
17          } catch (RuntimeException e) {
18              reply.writeException(e);
19              res = true;
20          } catch (OutOfMemoryError e) {
21              RuntimeException re = new RuntimeException("Out of memory", e);
22              reply.writeException(re);
23              res = true;
24          }
25          reply.recycle();
26          data.recycle();
27          return res;
28      }
29
30      ......
31 }
```

参数dataObj和replyObj分别指向C++层中的两个Parcel对象，其中，参数dataObj指向的Parcel对象包含了进程间通信数据，而参数replyObj指向的Parcel对象用来保存进程间通信结果数据。第6行和第7行首先将C++层中的这两个Parcel对象转换成Java层中的两个Parcel对象data和reply，接着第13行代码调用了由子类重写的成员函数onTransact来处理当前接收到的进程间通信请求。硬件访问服务FregService继承了IFregService.Stub类，并且继承了它的成员函数onTransact，因此，第13行代码实际上是调用了IFregService.Stub类的成员函数onTransact。

IFregService.Stub类的成员函数onTransact的实现如下所示。

out/target/common/obj/JAVA_LIBRARIES/framework_intermediates/src/core/java/android/os/IFregService.java

```
01 public interface IFregService extends android.os.IInterface {
02         /** Local-side IPC implementation stub class. */
03         public static abstract class Stub extends android.os.Binder implements android.os.IFregService {
04                 @Override
05                       public boolean onTransact(int code, android.os.Parcel data, android.
os.Parcel reply, int flags) throws android.os.RemoteException {
06                         switch (code) {
07                         ......
08                         case TRANSACTION_getVal: {
09                                 data.enforceInterface(DESCRIPTOR);
10                                 int _result = this.getVal();
11                                 reply.writeNoException();
12                                 reply.writeInt(_result);
13                                 return true;
14                         }
15                         }
16
17                         return super.onTransact(code, data, reply, flags);
18                 }
19         }
20
21         ......
22 }
```

由于应用程序Freg发送过来的进程间通信请求的类型为TRANSACTION_getVal，因此，第10行就会调用由子类实现的成员函数getVal来处理这个进程间通信请求。最后将处理结果保存在Parcel对象reply中，用来返回给应用程序Freg。

硬件访问服务FregService的成员函数getVal的实现如下所示。

frameworks/base/services/java/com/android/server/FregService.java

```
01 public class FregService extends IFregService.Stub {
02         ......
03
04         public int getVal() {
05                 if(mPtr == 0) {
06                         Slog.e(TAG, "Freg service is not initialized.");
07                         return 0;
08                 }
09
10                 return getVal_native(mPtr);
11         }
12
13         ......
14 }
```

第10行调用JNI方法getVal_native来读取虚拟硬件设备freg的寄存器val，最后将获得的值返回给调用者。JNI方法getVal_native的实现可以参考前面2.4小节的内容，这里不再详述。

至此，一个Java服务的调用过程就分析完成了，整个Binder进程间通信机制的Java接口也分析完成了。它们通过JNI方法来封装Binder库的C++接口，从而使得Android应用程序可以通过Java语言来使用Binder进程间通信机制。

第 6 章

Ashmem匿名共享内存系统

Android系统实现了一个匿名共享内存（Anonymous Shared Memory）子系统，用来在应用程序之间共享数据。匿名共享内存与传统的Linux系统实现的共享内存一样，都是基于内核提供的临时文件系统tmpfs实现的[1]，但是前者对内存块进行了更为精细化的管理。应用程序可以动态地将一块匿名共享内存划分为若干个小块，当这些小块内存不再需要使用时，它们就可以被内存管理系统回收。通过这种动态的、分而治之的内存管理方式，Android系统就能够有效地使用系统内存，适应内存较小的移动设备环境。

匿名共享内存系统是以Ashmem驱动程序为基础的，系统中所有的匿名共享内存都由Ashmem驱动程序负责分配和管理。为了方便应用程序使用匿名共享内存，Android系统在运行时库cutils中提供了C接口来访问Ashmem驱动程序；同时，又在应用程序框架层中提供了更高级的C++和Java接口来访问运行时库cutils中的匿名共享内存C接口。上述Ashmem匿名共享内存系统的架构如图6–1所示。

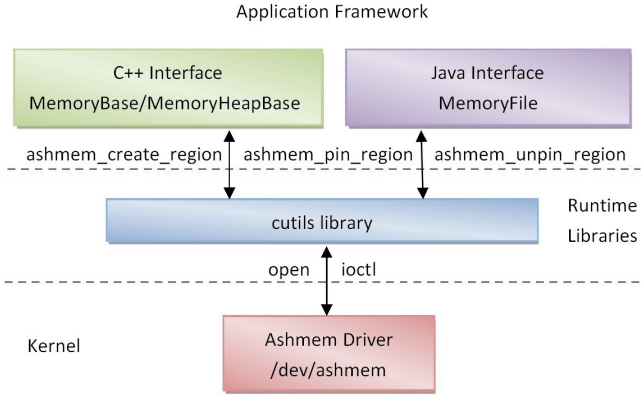


图6–1　Ashmem匿名共享内存系统架构

在应用程序框架层中，提供了两个C++类MemoryBase和MemoryHeapBase，以及一个Java类MemoryFile来使用匿名共享内存；而在运行时库cutils中，主要提供了三个C函数ashmem_create_region、ashmem_pin_region和ashmem_unpin_region来访问Ashmem驱动程序。Ashmem驱动程序在启动时，会创建一个/dev/ashmem设备文件，这样，运行时库cutils中的匿名共享内存C接口就可以通过文件操作函数open和ioctl等来访问Ashmem驱动程序。

传统的Linux系统使用一个整数来标志一块共享内存，而Android系统则使用一个文件描述符来标志一块匿名共享内存。使用文件描述符来描述一块匿名共享内存有两个好处：一是可以方便地将它映射到进程的地址空间，从而可以直接访问它的内容；二是可以使用Binder进程间通信机制来传输这个

1　传统的Linux系统的共享内存机制，可以参考《Linux内核源代码情景分析》一书的第6章第7节——共享内存。

文件描述符，从而实现在不同的应用程序之间共享一块匿名内存。Binder进程间通信机制使用一个类型为BINDER_TYPE_FD的Binder对象来描述一个文件描述符，当Binder驱动程序发现进程间通信数据中包含有这种Binder对象时，就会将对应的文件描述符复制到目标进程中，从而实现在两个进程中共享同一个文件。

接下来，我们就先分析Ashmem驱动程序的实现，接着再分析匿名共享内存系统在运行时库cutils中的C接口，以及在应用程序框架层中的C++和Java接口，最后结合Binder驱动程序来分析匿名共享内存在进程间的共享原理。

6.1 Ashmem驱动程序

Ashmem驱动程序实现在内核空间中，它的目录结构如下：

```
~/Android/kernel/goldfish
----include
    ----linux
        ----ashmem.h
----mm
    ----ashmem.c
```

它主要由ashmem.h和ashmem.c两个文件来实现。在本节中，我们首先介绍Ashmem驱动程序的基础数据结构，然后分析匿名共享内存设备的初始化、打开和内存映射过程，最后分析匿名共享内存块的锁定、解锁和回收过程。

6.1.1 基础数据结构

Ashmem驱动程序主要使用到了三个结构体：ashmem_area、ashmem_range和ashmem_pin，接下来就分别介绍它们的定义。

struct ashmem_area

kernel/goldfish/mm/ashmem.c

```
01 /*
02  * ashmem_area - anonymous shared memory area
03  * Lifecycle: From our parent file's open() until its release()
04  * Locking: Protected by 'ashmem_mutex'
05  * Big Note: Mappings do NOT pin this structure; it dies on close()
06  */
07 struct ashmem_area {
08         char name[ASHMEM_FULL_NAME_LEN];/* optional name for /proc/pid/maps */
09         struct list_head unpinned_list; /* list of all ashmem areas */
10         struct file *file;              /* the shmem-based backing file */
11         size_t size;                    /* size of the mapping, in bytes */
12         unsigned long prot_mask;        /* allowed prot bits, as vm_flags */
13 };
```

结构体ashmem_area用来描述一块匿名共享内存。

成员变量name用来保存一块匿名共享内存的名称，它同时会被写入到文件/proc/<pid>/maps中，其中，<pid>表示创建该块匿名共享内存的进程的PID。匿名共享内存名称的最大长度为ASHMEM_FULL_NAME_LEN，它的定义如下所示。

kernel/goldfish/mm/ashmem.c

```
1 #define ASHMEM_NAME_PREFIX "dev/ashmem/"
2 #define ASHMEM_NAME_PREFIX_LEN (sizeof(ASHMEM_NAME_PREFIX) - 1)
3 #define ASHMEM_FULL_NAME_LEN (ASHMEM_NAME_LEN + ASHMEM_NAME_PREFIX_LEN)
```

每一块匿名共享内存的名称都是以ASHMEM_NAME_PREFIX，即“dev/ashmem”为前缀的。应用程序在创建匿名共享内存时，如果没有指定名称，那么Ashmem驱动程序就会将它的名称指定为ASHMEM_NAME_DEF，即“dev/ashmem”，如下所示。

kernel/goldfish/include/linux/ashmem.h

```
1 #define ASHMEM_NAME_DEF         "dev/ashmem"
```

如果指定了名称，那么指定的名称的最大长度为ASHMEM_NAME_LEN，即256个字节，如下所示。

kernel/goldfish/include/linux/ashmem.h

```
1 #define ASHMEM_NAME_LEN         256
```

综上所述，一块匿名共享内存的全名称就等于前缀名称再加上应用程序指定的名称，它的最大长度等于ASHMEM_NAME_LEN的值加上ASHMEM_NAME_PREFIX_LEN的值，即等于ASHMEM_FULL_NAME_LEN。

成员变量unpinned_list用来描述一个解锁内存块列表。前面提到，一块匿名共享内存可以动态地划分为若干个小块，当这些小块的内存处于解锁状态时，它们就会被添加到所属匿名共享内存的解锁内存块列表中。解锁内存块列表中的内存块地址是互不相交的，并且按照地址值从大到小的顺序来排列。

成员变量file指向临时文件系统tmpfs中的一个文件，而成员变量size用来描述这个文件的大小。前面提到，匿名共享内存是基于Linux内核的临时文件系统tmpfs来实现的；因此，每一块匿名共享内存在临时文件系统tmpfs中都有一个对应的文件，这个文件的大小即为对应的匿名共享内存的大小。

成员变量prot_mask用来描述一块匿名共享内存的访问保护位。一块匿名共享内存的默认访问保护位被设置为PROT_MASK，它的定义如下所示。

kernel/goldfish/mm/ashmem.c

```
1 #define PROT_MASK               (PROT_EXEC | PROT_READ | PROT_WRITE)
```

PROT_EXEC、PROT_READ和PROT_WRITE分别表示可执行、可读和可写。

struct ashmem_range

kernel/goldfish/mm/ashmem.c

```
01 /*
02  * ashmem_range - represents an interval of unpinned (evictable) pages
03  * Lifecycle: From unpin to pin
04  * Locking: Protected by 'ashmem_mutex'
05  */
06 struct ashmem_range {
07         struct list_head lru;           /* entry in LRU list */
08         struct list_head unpinned;      /* entry in its area's unpinned list */
09         struct ashmem_area *asma;       /* associated area */
10         size_t pgstart;                 /* starting page, inclusive */
11         size_t pgend;                   /* ending page, inclusive */
12         unsigned int purged;            /* ASHMEM_NOT or ASHMEM_WAS_PURGED */
13 };
```

结构体ashmem_range用来描述一小块处于解锁状态的内存。

> **注意**
>
> 这些处于解锁状态的小块内存都是从一块匿名共享内存中划分出来的，它们通过成员变量unpinned链入到宿主匿名共享内存的解锁内存块列表unpinned_list中。一块处于解锁状态的内存的宿主匿名共享内存是通过成员变量asma来描述的。

在Ashmem驱动程序中，每一块处于解锁状态的内存还会通过其成员变量lru链入到一个全局列表

ashmem_lru_list中。全局列表ashmem_lru_list是一个最近最少使用列表（Least Recently Used），它的定义如下所示。

kernel/goldfish/mm/ashmem.c

```
1 /* LRU list of unpinned pages, protected by ashmem_mutex */
2 static LIST_HEAD(ashmem_lru_list);
```

由于处于解锁状态的内存都是不再需要使用的，因此，当系统内存不足时，内存管理系统就会按照最近最少使用的原则来回收保存在全局列表ashmem_lru_list中的内存块。

成员变量pgstart和pgend分别用来描述一块处于解锁状态的内存的开始地址和结束地址，它们的单位是页。

最后，成员变量purged用来描述一块处于解锁状态的内存是否已经被回收。如果是，那么它的值就等于ASHMEM_WAS_PURGED；否则，就等于ASHMEM_NOT_PURGED。ASHMEM_NOT_PURGED和ASHMEM_WAS_PURGED是两个宏，它们的定义如下所示。

kernel/goldfish/include/linux/ashmem.h

```
1 #define ASHMEM_NOT_PURGED      0
2 #define ASHMEM_WAS_PURGED      1
```

Ashmem驱动程序中的所有ashmem_area结构体和ashmem_range结构体的访问都是受互斥锁ashmem_mutex保护的，它的定义如下所示。

kernel/goldfish/mm/ashmem.c

```
1 static DEFINE_MUTEX(ashmem_mutex);
```

struct ashmem_pin

kernel/goldfish/include/linux/ashmem.h

```
1 struct ashmem_pin {
2         __u32 offset;   /* offset into region, in bytes, page-aligned */
3         __u32 len;      /* length forward from offset, in bytes, page-aligned */
4 };
```

结构体ashmem_pin是Ashmem驱动程序定义的IO控制命令ASHMEM_PIN和ASHMEM_UNPIN的参数，用来描述一小块即将被锁定或者解锁的内存。其中，成员变量offset表示这块即将被锁定或者解锁的内存在其宿主匿名共享内存块中的偏移值，而成员变量len表示要被锁定或者解锁的内存块的大小，它们都是以字节为单位，并且对齐到页面边界的。

6.1.2 匿名共享内存设备的初始化过程

匿名共享内存设备是在Ashmem驱动程序的启动函数ashmem_init中初始化的，它的定义如下所示。

kernel/goldfish/mm/ashmem.c

```
01 static int __init ashmem_init(void)
02 {
03         int ret;
04
05         ashmem_area_cachep = kmem_cache_create("ashmem_area_cache",
06                                                sizeof(struct ashmem_area),
07                                                0, 0, NULL);
08         if (unlikely(!ashmem_area_cachep)) {
09                 printk(KERN_ERR "ashmem: failed to create slab cache\n");
10                 return -ENOMEM;
11         }
12
13         ashmem_range_cachep = kmem_cache_create("ashmem_range_cache",
14                                                sizeof(struct ashmem_range),
```

```
15                                          0, 0, NULL);
16          if (unlikely(!ashmem_range_cachep)) {
17                  printk(KERN_ERR "ashmem: failed to create slab cache\n");
18                  return -ENOMEM;
19          }
20
21          ret = misc_register(&ashmem_misc);
22          if (unlikely(ret)) {
23                  printk(KERN_ERR "ashmem: failed to register misc device!\n");
24                  return ret;
25          }
26
27          register_shrinker(&ashmem_shrinker);
28
29          printk(KERN_INFO "ashmem: initialized\n");
30
31          return 0;
32  }
```

除第21行调用函数misc_register注册一个匿名共享内存设备之外，第5行和第13行还分别调用函数kmem_cache_create创建了两个slab缓冲区分配器ashmem_area_cachep和ashmem_range_cachep[1]；其中，前者用来分配ashmem_area结构体，后者用来分配ashmem_range结构体。

slab缓冲区分配器ashmem_area_cachep和ashmem_range_cachep的定义如下所示。

kernel/goldfish/mm/ashmem.c

```
1 static struct kmem_cache *ashmem_area_cachep __read_mostly;
2 static struct kmem_cache *ashmem_range_cachep __read_mostly;
```

匿名共享内存设备是一个类型为misc的字符设备，它是由miscdevice 结构体ashmem_misc来定义的，如下所示。

kernel/goldfish/mm/ashmem.c

```
1 static struct miscdevice ashmem_misc = {
2          .minor = MISC_DYNAMIC_MINOR,
3          .name = "ashmem",
4          .fops = &ashmem_fops,
5 };
```

类型为misc的字符设备的介绍，以及它们的注册函数misc_register的实现可以参考4.1节的内容。从miscdevice 结构体ashmem_misc的定义可以知道，匿名共享内存设备对应的设备文件为/dev/ashmem，应用程序可以通过它来访问Ashmem驱动程序。

设备文件/dev/ashmem的操作方法列表由file_operations结构体ashmem_fops来描述，它的定义如下所示。

kernel/goldfish/mm/ashmem.c

```
1 static struct file_operations ashmem_fops = {
2          .owner = THIS_MODULE,
3          .open = ashmem_open,
4          .release = ashmem_release,
5          .mmap = ashmem_mmap,
6          .unlocked_ioctl = ashmem_ioctl,
7          .compat_ioctl = ashmem_ioctl,
8 };
```

从file_operations结构体ashmem_fops的定义可以知道，设备文件/dev/ashmem对应的文件打开、关闭、内存映射、IO控制函数分别为ashmem_open、ashmem_release、ashmem_mmap和ashmem_ioctl。由

1　Linux内核采用slab机制来管理内核中小对象的内存分配和释放，具体可以参考*Understanding the Linux Kernel*一书的第8章——Memory Management。

于匿名共享内存的访问方式是直接地址访问，即先映射进程的地址空间，然后再通过虚拟地址来直接访问，因此，设备文件/dev/ashmem就没有对应的读写函数。

第27行调用函数register_shrinker向内存管理系统注册一个内存回收函数ashmem_shrinker。当系统内存不足时，内存管理系统就会通过一个页框回收算法（Page Frame Reclaiming Algorithm，PFRA）来回收内存，这时候所有调用函数register_shrinker注册的内存回收函数都会被调用[1]，以便它们可以向系统贡献空闲或者相对空闲的内存。

6.1.3 匿名共享内存设备文件的打开过程

当应用程序调用函数open打开设备文件/dev/ashmem时，Ashmem驱动程序中的函数ashmem_open就会被调用，它主要用来为应用程序创建一个ashmem_area结构体，它的实现如下所示。

kernel/goldfish/mm/ashmem.c

```
01 static int ashmem_open(struct inode *inode, struct file *file)
02 {
03         struct ashmem_area *asma;
04         int ret;
05
06         ret = nonseekable_open(inode, file);
07         if (unlikely(ret))
08                 return ret;
09
10         asma = kmem_cache_zalloc(ashmem_area_cachep, GFP_KERNEL);
11         if (unlikely(!asma))
12                 return -ENOMEM;
13
14         INIT_LIST_HEAD(&asma->unpinned_list);
15         memcpy(asma->name, ASHMEM_NAME_PREFIX, ASHMEM_NAME_PREFIX_LEN);
16         asma->prot_mask = PROT_MASK;
17         file->private_data = asma;
18
19         return 0;
20 }
```

第6行调用函数nonseekable_open将设备文件/dev/ashmem设置为不可随机访问。这是由于设备文件/dev/ashmem只是一个用来访问Ashmem驱动程序的通道，它本身并没有内容，因此，对它进行随机访问是没有意义的。

第10行调用函数kmem_cache_zalloc从slab缓冲区ashmem_area_cachep中分配一个ashmem_area结构体asma，并且第17行将它设置到参数file的成员变量private_data中。第14行初始化ashmem_area结构体asma的解锁内存块列表unpinned_list，接着第15行和第16行分别设置ashmem_area结构体asma所描述的匿名共享内存块的名称和访问保护位。

从第15行代码可以知道，每一块匿名共享内存在创建时，它的名称都被指定为ASHMEM_NAME_PREFIX。我们可以通过IO控制命令ASHMEM_SET_NAME来修改匿名共享内存块的名称。

IO控制命令ASHMEM_SET_NAME的定义如下所示。

```
kernel/goldfish/include/linux/ashmem.h
1 #define ASHMEM_SET_NAME         _IOW(__ASHMEMIOC, 1, char[ASHMEM_NAME_LEN])
```

它后面跟的参数是一个字符数组，最大长度为ASHMEM_NAME_LEN 。

IO控制命令ASHMEM_SET_NAME是由Ashmem驱动程序中的函数ashmem_ioctl来处理的，如下所示。

1 页框回收算法可以参考*Understanding the Linux Kernel*一书的第17章——Page Frame Reclaiming。

kernel/goldfish/mm/ashmem.c

```
01 static long ashmem_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
02 {
03         struct ashmem_area *asma = file->private_data;
04         long ret = -ENOTTY;
05
06         switch (cmd) {
07         case ASHMEM_SET_NAME:
08                 ret = set_name(asma, (void __user *) arg);
09                 break;
10         ......
11         }
12
13         return ret;
14 }
```

第3行获得用来描述要修改名称的匿名共享内存asma，接着第8行调用函数set_name来修改它的名称。

函数set_name的实现如下所示。

kernel/goldfish/mm/ashmem.c

```
01 static int set_name(struct ashmem_area *asma, void __user *name)
02 {
03         int ret = 0;
04
05         mutex_lock(&ashmem_mutex);
06
07         /* cannot change an existing mapping's name */
08         if (unlikely(asma->file)) {
09                 ret = -EINVAL;
10                 goto out;
11         }
12
13         if (unlikely(copy_from_user(asma->name + ASHMEM_NAME_PREFIX_LEN,
14                                     name, ASHMEM_NAME_LEN)))
15                 ret = -EFAULT;
16         asma->name[ASHMEM_FULL_NAME_LEN-1] = '\0';
17
18 out:
19         mutex_unlock(&ashmem_mutex);
20
21         return ret;
22 }
```

前面提到，每一块匿名共享内存在临时文件系统tmpfs中都对应有一个文件，这个临时文件的名称就等于与它对应的匿名共享内存的名称。当Ashmem驱动程序为一块匿名共享内存创建了临时文件之后，应用程序就不可以再修改它的名称了。因此，在修改匿名共享内存asma的名称之前，首先要检查Ashmem驱动程序是否已经为它创建过临时文件了，即它的成员变量file是否已经指向一个打开文件结构体。如果是，即第8行的if语句为true，那么第10行就直接返回；否则，第13行就将应用程序指定的名称复制到匿名共享内存asma的前缀名称的后面，最后得到匿名共享内存asma的全名称的形式就为ASHMEM_NAME_PREFIX<name>，即/dev/ashmem/<name>，其中，<name>为应用程序指定的名称。

应用程序除了可以设置匿名共享内存的名称，还可以设置它的大小，这是通过IO控制命令ASHMEM_SET_SIZE来实现的。

IO控制命令ASHMEM_SET_SIZE的定义如下所示。

kernel/goldfish/include/linux/ashmem.h

```
1 #define ASHMEM_SET_SIZE         _IOW(__ASHMEMIOC, 3, size_t)
```

它后面跟的参数是一个无符号整数。

IO控制命令ASHMEM_SET_SIZE同样是由Ashmem驱动程序中的函数ashmem_ioctl来处理的，如下所示。

kernel/goldfish/mm/ashmem.c

```
01 static long ashmem_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
02 {
03         struct ashmem_area *asma = file->private_data;
04         long ret = -ENOTTY;
05
06         switch (cmd) {
07         ......
08         case ASHMEM_SET_SIZE:
09                 ret = -EINVAL;
10                 if (!asma->file) {
11                         ret = 0;
12                         asma->size = (size_t) arg;
13                 }
14                 break;
15         ......
16         }
17
18         return ret;
19 }
```

第3行获得用来描述要修改大小的匿名共享内存asma，接着第10行判断Ashmem驱动程序是否已经为匿名共享内存asma创建过临时文件了。如果是，函数什么也不做就直接返回；否则，第12行就将它的大小修改为指定值。

从IO控制命令ASHMEM_SET_NAME和ASHMEM_SET_SIZE的处理过程就可以看出，如果应用程序要修改一块匿名共享内存的大小和名称，就必须要在Ashmem驱动程序为它创建临时文件之前进行。

当应用程序调用函数mmap将前面打开的设备文件/dev/ashmem映射到进程的地址空间时，Ashmem驱动程序就会为一块匿名共享内存创建一个临时文件。下面我们继续分析设备文件/dev/ashmem的内存映射过程。

6.1.4 匿名共享内存设备文件的内存映射过程

当应用程序调用函数mmap将前面打开的设备文件/dev/ashmem映射到进程的地址空间时，Ashmem驱动程序中的函数ashmem_mmap就会被执行，用来为前面创建的一块匿名共享内存创建一个临时文件。

函数ashmem_mmap的实现如下所示。

kernel/goldfish/mm/ashmem.c

```
01 static int ashmem_mmap(struct file *file, struct vm_area_struct *vma)
02 {
03         struct ashmem_area *asma = file->private_data;
04         int ret = 0;
05
06         mutex_lock(&ashmem_mutex);
07
08         /* user needs to SET_SIZE before mapping */
09         if (unlikely(!asma->size)) {
10                 ret = -EINVAL;
11                 goto out;
12         }
13
14         /* requested protection bits must match our allowed protection mask */
15         if (unlikely((vma->vm_flags & ~asma->prot_mask) & PROT_MASK)) {
```

```
16                  ret = -EPERM;
17                  goto out;
18          }
19
20          if (!asma->file) {
21                  char *name = ASHMEM_NAME_DEF;
22                  struct file *vmfile;
23
24                  if (asma->name[ASHMEM_NAME_PREFIX_LEN] != '\0')
25                          name = asma->name;
26
27                  /* ... and allocate the backing shmem file */
28                  vmfile = shmem_file_setup(name, asma->size, vma->vm_flags);
29                  if (unlikely(IS_ERR(vmfile))) {
30                          ret = PTR_ERR(vmfile);
31                          goto out;
32                  }
33                  asma->file = vmfile;
34          }
35          get_file(asma->file);
36
37          if (vma->vm_flags & VM_SHARED)
38                  shmem_set_file(vma, asma->file);
39          else {
40                  if (vma->vm_file)
41                          fput(vma->vm_file);
42                  vma->vm_file = asma->file;
43          }
44          vma->vm_flags |= VM_CAN_NONLINEAR;
45
46 out:
47          mutex_unlock(&ashmem_mutex);
48          return ret;
49 }
```

第9行的if语句检查匿名共享内存asma的大小是否等于0。如果等于0，那么第10行和第11行就直接出错返回了。因此，应用程序在将设备文件/dev/ashmem映射到进程的地址空间之前，应该先使用IO控制命令ASHMEM_SET_SIZE来设置要创建的匿名共享内存的大小。

第15行的if语句检查要映射的虚拟内存vma的访问保护权限是否超出了匿名共享内存asma的访问保护权限prot_mask。如果是，那么第16行和第17行也直接出错返回了。例如，如果匿名共享内存asma只允许执行读操作，而虚拟内存vma除了允许读操作，还允许写操作，那么这时候第15行的if语句就为true了。因此，第16行和第17行就会出错返回。

第20行的if语句检查之前是否已经为匿名共享内存asma创建过临时文件。如果没有创建过，接下来第21行到第33行代码就会为它创建一个临时文件。第21行首先将要创建的临时文件的名称设置为ASHMEM_NAME_DEF，即“/dev/ashmem”，然后第24行检查应用程序是否设置过匿名共享内存asma的名称。如果设置过，第25行就将它的名称作为即将要创建的临时文件的名称。接着第28行就会调用函数shmem_file_setup在临时文件系统tmpfs中创建一个临时文件，并且将得到的打开文件结构体保存在匿名共享内存asma的成员变量file中。

第37行的if语句检查要映射的虚拟内存vma是否允许在不同的进程之间共享。如果允许，第38行就调用函数shmem_set_file设置它的映射文件，以及它的内存操作方法表。

函数shmem_set_file的实现如下所示。

kernel/goldfish/mm/shmem.c

```
1 void shmem_set_file(struct vm_area_struct *vma, struct file *file)
2 {
3         if (vma->vm_file)
```

```
4                   fput(vma->vm_file);
5           vma->vm_file = file;
6           vma->vm_ops = &shmem_vm_ops;
7 }
```

第5行将虚拟内存vma的映射文件设置为前面为匿名共享内存asma所创建的临时文件，接着第6行将虚拟内存vma的内存操作方法表设置为shmem_vm_ops。shmem_vm_ops是一个类型为vm_operations_struct的结构体，它的成员变量fault指向了函数shmem_fault。

开始的时候，虚拟内存vma是没有映射物理页面的；因此，当它第一次被访问时，就会发生缺页异常（Page Fault），这时候内核就会调用它的内存操作方法表中的函数shmem_fault给它映射物理页面。函数shmem_fault首先会在页面缓冲区中检查是否存在与缺页的虚拟地址对应的物理页面。如果存在，就直接将它们映射到缺页的虚拟地址；否则，再去页面换出设备中检查是否存在与缺页的虚拟地址对应的换出页面。如果存在，就先把它们添加到页面缓冲区中，然后再映射到缺页的虚拟地址；否则，就需要为缺页的虚拟地址分配新的物理页面，并且从虚拟内存vma的映射文件vm_file中读入相应的内容来初始化这些新分配的物理页面，最后将这些物理页面加入到页面缓冲区中去[1]。通过这种方式，我们就可以将一个物理页面映射到两个不同进程的虚拟地址空间，从而通过内存映射机制来实现共享内存的功能。

回到函数ashmem_mmap中，如果要映射的虚拟内存vma不允许在不同的进程之间共享，即第37行的if语句为false；那么第40行到第42行代码就只将前面创建的临时文件设置为要映射的虚拟内存vma的映射文件，而不会设置它的内存操作方法表。当访问虚拟内存vma发生缺页异常时，内核就会直接为缺页的地址分配新的物理页面，这样，就阻止了虚拟内存vma的内容被其他进程共享了。

至此，一块匿名共享内存就创建完成了。接下来，我们继续分析Ashmem驱动程序是如何通过分块的形式来管理匿名共享内存的。

6.1.5 匿名共享内存块的锁定和解锁过程

匿名共享内存是以分块的形式来管理的，应用程序可以对这些小块内存执行锁定或者解锁操作，其中，处于解锁状态的内存是可以被内存管理系统回收的。

一块匿名共享内存在创建时是处于锁定状态的，接下来，应用程序可以根据需要把它划分成若干个小块来使用。当其中的某些小块内存不再使用时，应用程序就可以对它们执行解锁操作，从而可以在内存紧张时为内存管理系统贡献内存。处于解锁状态的内存如果还没有被回收，那么应用程序还可以对它们执行锁定操作，从而阻止它们被内存管理系统回收。

Ashmem驱动程序分别提供两个IO控制命令ASHMEM_PIN和ASHMEM_UNPIN来锁定和解锁一块匿名共享内存，它们的定义如下所示。

kernel/goldfish/include/linux/ashmem.h

```
1 #define ASHMEM_PIN              _IOW(__ASHMEMIOC, 7, struct ashmem_pin)
2 #define ASHMEM_UNPIN            _IOW(__ASHMEMIOC, 8, struct ashmem_pin)
```

IO控制命令ASHMEM_PIN和ASHMEM_UNPIN后面跟的参数均为一个ashmem_pin结构体，它们是由Ashmem驱动程序中的函数ashmem_ioctl负责处理的，如下所示。

kernel/goldfish/mm/ashmem.c

```
01 static long ashmem_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
02 {
```

1 函数shmem_fault的实现原理可以参考《Linux内核源代码情景分析》一书的第6章第7节——共享内存，或者*Understanding the Linux Kernel*一书的第1章第3节第5部分——IPC Shared Memory。

```
03          struct ashmem_area *asma = file->private_data;
04          long ret = -ENOTTY;
05
06          switch (cmd) {
07          ......
08          case ASHMEM_PIN:
09          case ASHMEM_UNPIN:
10          ......
11                  ret = ashmem_pin_unpin(asma, cmd, (void __user *) arg);
12                  break;
13          ......
14          }
15
16          return ret;
17  }
```

第3行首先获得要执行内存块锁定或者解锁的匿名共享内存asma，接着第11行将这两个IO控制命令分发给函数ashmem_pin_unpin来处理。

函数ashmem_pin_unpin的实现如下所示。

kernel/goldfish/mm/ashmem.c

```
01 static int ashmem_pin_unpin(struct ashmem_area *asma, unsigned long cmd,
02                              void __user *p)
03 {
04          struct ashmem_pin pin;
05          size_t pgstart, pgend;
06          int ret = -EINVAL;
07
08          if (unlikely(!asma->file))
09                  return -EINVAL;
10
11          if (unlikely(copy_from_user(&pin, p, sizeof(pin))))
12                  return -EFAULT;
13
14          /* per custom, you can pass zero for len to mean "everything onward" */
15          if (!pin.len)
16                  pin.len = PAGE_ALIGN(asma->size) - pin.offset;
17
18          if (unlikely((pin.offset | pin.len) & ~PAGE_MASK))
19                  return -EINVAL;
20
21          if (unlikely(((__u32) -1) - pin.offset < pin.len))
22                  return -EINVAL;
23
24          if (unlikely(PAGE_ALIGN(asma->size) < pin.offset + pin.len))
25                  return -EINVAL;
26
27          pgstart = pin.offset / PAGE_SIZE;
28          pgend = pgstart + (pin.len / PAGE_SIZE) - 1;
29
30          mutex_lock(&ashmem_mutex);
31
32          switch (cmd) {
33          case ASHMEM_PIN:
34                  ret = ashmem_pin(asma, pgstart, pgend);
35                  break;
36          case ASHMEM_UNPIN:
37                  ret = ashmem_unpin(asma, pgstart, pgend);
38                  break;
39          ......
40          }
41
```

```
42          mutex_unlock(&ashmem_mutex);
43
44          return ret;
45  }
```

第8行的if语句检查是否已经为匿名共享内存asma创建过临时文件。如果没有创建过，第9行就直接出错返回。

第11行调用函数copy_from_user将用户空间传进来的参数复制到一个ashmem_pin结构体pin中，这样函数接下来就可以通过这个结构体来获得要锁定或者解锁的内存块的起始位置和大小了。

第15行的if语句检查锁定或者解锁的内存块的大小是否等于0。如果等于0，就意味着要锁定或者解锁的内存块从ashmem_pin结构体pin所描述的偏移地址开始，一直到匿名共享内存asma的末尾。

第18行的if语句检查要锁定或者解锁的内存块的偏移地址和大小是否对齐到页面边界。如果不是，第19行就直接出错返回。

第21行的if语句检查要锁定或者解锁的内存块的末尾地址是否超出一个无符号整数。如果是，第22行就直接出错返回。

第24行的if语句检查要锁定或者解锁的内存块的末尾地址是否超出匿名共享内存asma的末尾地址。如果是，第23就直接出错返回。

通过了前面的合法检查之后，第27行和第28行代码计算要锁定或者解锁的内存块的开始地址和结束地址，并且保存在变量pgstart和pgend中。它们是以页为单位的，并且是一个相对地址，即相对匿名共享内存asma的起始地址。

如果要执行的操作是锁定一块共享内存，那么第34行就调用函数ashmem_pin来继续处理；否则，要执行的操作就是解锁一块共享内存，因此，第37行就调用函数ashmem_unpin来继续处理。接下来，我们就分别分析函数ashmem_unpin和ashmem_pin的实现。

函数ashmem_unpin是用来解锁一块匿名共享内存的，它的实现如下所示。

kernel/goldfish/mm/ashmem.c

```
01 /*
02  * ashmem_unpin - unpin the given range of pages. Returns zero on success.
03  *
04  * Caller must hold ashmem_mutex.
05  */
06 static int ashmem_unpin(struct ashmem_area *asma, size_t pgstart, size_t pgend)
07 {
08          struct ashmem_range *range, *next;
09          unsigned int purged = ASHMEM_NOT_PURGED;
10
11 restart:
12          list_for_each_entry_safe(range, next, &asma->unpinned_list, unpinned) {
13                  /* short circuit: this is our insertion point */
14                  if (range_before_page(range, pgstart))
15                          break;
16
17                  /*
18                   * The user can ask us to unpin pages that are already entirely
19                   * or partially pinned. We handle those two cases here.
20                   */
21                  if (page_range_subsumed_by_range(range, pgstart, pgend))
22                          return 0;
23                  if (page_range_in_range(range, pgstart, pgend)) {
24                          pgstart = min_t(size_t, range->pgstart, pgstart),
25                          pgend = max_t(size_t, range->pgend, pgend);
26                          purged |= range->purged;
27                          range_del(range);
28                          goto restart;
```

```
29                  }
30          }
31
32          return range_alloc(asma, range, purged, pgstart, pgend);
33  }
```

前面提到，一块匿名共享内存的所有解锁内存块都是按照地址值从大到小的顺序保存在其解锁内存块列表unpinned_list中的，并且它们是互不相交的。因此，函数ashmem_unpin在解锁一个内存块时，首先检查这个内存块是否与那些处于解锁状态的内存块相交。如果相交，就要对它们进行合并，然后在目标匿名共享内存的解锁内存块列表unpinned_list中找到一个合适的位置来保存合并后得到的内存块。

第12行到第30行的循环在目标匿名共享内存asma的解锁内存块列表unpinned_list中遍历每一块处于解锁状态的内存。如果发现一个已解锁内存块range与即将要解锁的内存块[pgstart, pgend]相交，那么就需要对它们执行合并操作，即调整参数pgstart和pgend的值，使得已解锁内存块range包含在它里面。

已解锁内存块range与即将要解锁的内存块[pgstart, pgend]的位置关系如图6-2中的A、B、C、D和E示意图所示。

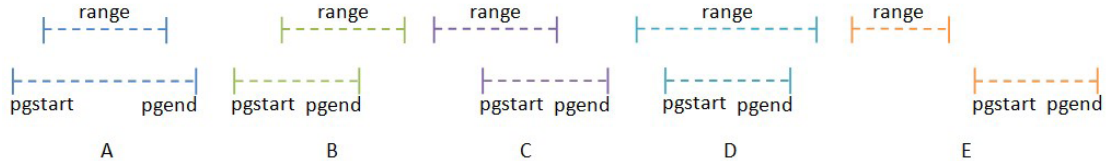


图6-2　已解锁内存块range与即将要解锁的内存块[pgstart, pgend]的位置关系

在A、B和C三个示意图中，函数ashmem_unpin需要合并已解锁内存块range与即将要解锁的内存块[pgstart, pgend]，即调整参数pgstart和pgend的值，如第24行和第25行代码所示。由于调整了参数pgstart和pgend的值之后，即将要解锁的内存块[pgstart, pgend]就包含已解锁内存块range了，因此，第27行调用函数range_del将后者从匿名共享内存asma的解锁内存块列表unpinned_list中删除。此外，合并后得到的内存块[pgstart, pgend]可能还会与其他的已解锁内存块相交，因此，第28行需要重新执行第12行的for循环来执行下一个合并操作。

在D示意图中，由于已解锁内存块range已经包含了即将要解锁的内存块[pgstart, pgend]，因此，函数ashmem_unpin不需要做任何处理就可以返回，如第21行和第22行代码所示。

在E示意图中，由于已解锁内存块range与即将要解锁的内存块[pgstart, pgend]不相交，因此，函数ashmem_unpin不需要对它们进行合并。此外，由于已解锁内存块range的地址区间小于即将要解锁的内存块[pgstart, pgend]的地址区间，因此，函数ashmem_unpin可以停止遍历目标匿名共享内存的解锁内存块列表unpinned_list，如第14行和第15行代码所示。

第12行到第30行的循环执行完成之后，如果没有出现示意图D所示的情况，那么第32行就会调用函数range_alloc将即将要解锁的内存块[pgstart, pgend]插入到目标匿名共享内存asma的解锁内存块列表unpinned_list中，插入的位置就在已解锁内存块range的前面。

Ashmem驱动程序定义了五个宏：page_range_subsumes_range 、page_range_subsumed_by_range 、page_in_range 、page_range_in_range 和range_before_page，它们用来判断两个内存块的位置关系，如下所示。

kernel/goldfish/mm/ashmem.c

```
01 #define page_range_subsumes_range(range, start, end) \
02   (((range)->pgstart >= (start)) && ((range)->pgend <= (end)))
03
```

```
04 #define page_range_subsumed_by_range(range, start, end) \
05   (((range)->pgstart <= (start)) && ((range)->pgend >= (end)))
06
07 #define page_in_range(range, page) \
08  (((range)->pgstart <= (page)) && ((range)->pgend >= (page)))
09
10 #define page_range_in_range(range, start, end) \
11   (page_in_range(range, start) || page_in_range(range, end) || \
12    page_range_subsumes_range(range, start, end))
13
14 #define range_before_page(range, page) \
15   ((range)->pgend < (page))
```

宏page_range_subsumes_range用来判断内存块[start, end]是否包含内存块range。

宏page_range_subsumed_by_range用来判断内存块range是否包含内存块[start, end]。

宏page_in_range用来判断内存块地址page是否包含在内存块range中。

宏page_range_in_range用来判断内存块[start, end]是否互相包含或者相交。

宏range_before_page用来判断内存块地址page是否大于内存块range的末尾地址。

回到函数ashmem_unpin中，接下来我们继续分析函数range_del和range_alloc是如何从目标匿名共享内存asma中删除和添加一块处于解锁状态的内存块的。

函数range_del的实现如下所示。

kernel/goldfish/mm/ashmem.c

```
1 static void range_del(struct ashmem_range *range)
2 {
3         list_del(&range->unpinned);
4         if (range_on_lru(range))
5                 lru_del(range);
6         kmem_cache_free(ashmem_range_cachep, range);
7 }
```

首先第3行调用函数list_del将解锁内存块range从其宿主匿名共享内块的解锁内存块列表unpinned_list中删除。接着第4行判断内存块range是否位于全局列表ashmem_lru_list中。如果是，那么第5行就调用函数lru_del将它从全局列表ashmem_lru_list中删除。最后第6行将ashmem_range结构体range所占用的内存交回到slab缓冲区ashmem_range_cachep中。

判断一个处于解锁状态的内存块是否位于全局列表ashmem_lru_list中是通过宏range_on_lru来实现的，它的定义如下所示。

kernel/goldfish/mm/ashmem.c

```
1 #define range_on_lru(range) \
2   ((range)->purged == ASHMEM_NOT_PURGED)
```

如果处于解锁状态的内存块range的成员变量purged的值等于ASHMEM_NOT_PURGED，那么就说明它还没有被内存管理系统回收，这时候它就一定位于全局列表ashmem_lru_list中。

函数lru_del用来将一块处于解锁状态的内存块从全局列表ashmem_lru_list中删除，它的实现如下所示。

kernel/goldfish/mm/ashmem.c

```
1 static inline void lru_del(struct ashmem_range *range)
2 {
3         list_del(&range->lru);
4         lru_count -= range_size(range);
5 }
```

首先第3行调用函数list_del将处于解锁状态的内存块range从全局列表ashmem_lru_list中删除，接

着第4行先计算处于解锁状态的内存块range的大小，然后再将它从全局变量lru_count中减去。

Ashmem驱动程序使用全局变量lru_count来维护保存在全局列表ashmem_lru_list中处于解锁状态的内存块的总大小，用来表示可以被内存管理系统回收的匿名共享内存的容量，它是以页为单位的。

宏range_size用来计算一个处于解锁状态的内存块的大小，它的定义如下所示。

kernel/goldfish/mm/ashmem.c

```
1 #define range_size(range) \
2   ((range)->pgend - (range)->pgstart + 1)
```

将处于解锁状态的内存块range的末尾地址减去开始地址，再加上1个页面，就可以得到它的大小。之所以要再加上1个页面，是因为range->pgstart和range->pgend描述的是一个闭合的地址区间。

函数range_alloc的实现如下所示。

kernel/goldfish/mm/ashmem.c

```
01 /*
02  * range_alloc - allocate and initialize a new ashmem_range structure
03  *
04  * 'asma' - associated ashmem_area
05  * 'prev_range' - the previous ashmem_range in the sorted asma->unpinned list
06  * 'purged' - initial purge value (ASMEM_NOT_PURGED or ASHMEM_WAS_PURGED)
07  * 'start' - starting page, inclusive
08  * 'end' - ending page, inclusive
09  *
10  * Caller must hold ashmem_mutex.
11  */
12 static int range_alloc(struct ashmem_area *asma,
13                        struct ashmem_range *prev_range, unsigned int purged,
14                        size_t start, size_t end)
15 {
16         struct ashmem_range *range;
17
18         range = kmem_cache_zalloc(ashmem_range_cachep, GFP_KERNEL);
19         if (unlikely(!range))
20                 return -ENOMEM;
21
22         range->asma = asma;
23         range->pgstart = start;
24         range->pgend = end;
25         range->purged = purged;
26
27         list_add_tail(&range->unpinned, &prev_range->unpinned);
28
29         if (range_on_lru(range))
30                 lru_add(range);
31
32         return 0;
33 }
```

第18行首先调用函数kmem_cache_zalloc从slab缓冲区ashmem_range_cachep中分配一个ashmem_range结构体range，用来描述即将要解锁的内存块[start, end]。

第22行到第25行代码对ashmem_range结构体range进行初始化，接着第27行调用函数list_add_tail将它插入到其宿主匿名共享内存的解锁内存块列表unpinned_list中，插入的位置就在其宿主匿名共享内存的另外一块处于解锁状态的内存块prev_range的前面。

最后，第29行的if语句判断已解锁的内存块[start, end]是否已经被内存系统回收。如果没有，那么

第30行就调用函数lru_add将它添加到全局列表ashmem_lru_list中。

函数lru_add的实现如下所示。

kernel/goldfish/mm/ashmem.c

```
1 static inline void lru_add(struct ashmem_range *range)
2 {
3         list_add_tail(&range->lru, &ashmem_lru_list);
4         lru_count += range_size(range);
5 }
```

首先第3行调用函数list_add_tail将已解锁的内存块range添加到全局列表ashmem_lru_list中，接着第4行先调用宏range_size来计算已解锁的内存块range的大小，然后再将它增加到全局变量lru_count中，表示目前可以被内存管理系统回收的匿名共享内存的容量。

函数ashmem_pin是用来锁定一块匿名共享内存的，它的实现如下所示。

kernel/goldfish/mm/ashmem.c

```
01 /*
02  * ashmem_pin - pin the given ashmem region, returning whether it was
03  * previously purged (ASHMEM_WAS_PURGED) or not (ASHMEM_NOT_PURGED).
04  *
05  * Caller must hold ashmem_mutex.
06  */
07 static int ashmem_pin(struct ashmem_area *asma, size_t pgstart, size_t pgend)
08 {
09         struct ashmem_range *range, *next;
10         int ret = ASHMEM_NOT_PURGED;
11
12         list_for_each_entry_safe(range, next, &asma->unpinned_list, unpinned) {
13                 /* moved past last applicable page; we can short circuit */
14                 if (range_before_page(range, pgstart))
15                         break;
16
17                 /*
18                  * The user can ask us to pin pages that span multiple ranges,
19                  * or to pin pages that aren't even unpinned, so this is messy.
20                  *
21                  * Four cases:
22                  * 1. The requested range subsumes an existing range, so we
23                  *    just remove the entire matching range.
24                  * 2. The requested range overlaps the start of an existing
25                  *    range, so we just update that range.
26                  * 3. The requested range overlaps the end of an existing
27                  *    range, so we just update that range.
28                  * 4. The requested range punches a hole in an existing range,
29                  *    so we have to update one side of the range and then
30                  *    create a new range for the other side.
31                  */
32                 if (page_range_in_range(range, pgstart, pgend)) {
33                         ret |= range->purged;
34
35                         /* Case #1: Easy. Just nuke the whole thing. */
36                         if (page_range_subsumes_range(range, pgstart, pgend)) {
37                                 range_del(range);
38                                 continue;
39                         }
40
41                         /* Case #2: We overlap from the start, so adjust it */
42                         if (range->pgstart >= pgstart) {
43                                 range_shrink(range, pgend + 1, range->pgend);
44                                 continue;
45                         }
```

```
46
47                          /* Case #3: We overlap from the rear, so adjust it */
48                          if (range->pgend <= pgend) {
49                                  range_shrink(range, range->pgstart, pgstart-1);
50                                  continue;
51                          }
52
53                          /*
54                           * Case #4: We eat a chunk out of the middle. A bit
55                           * more complicated, we allocate a new range for the
56                           * second half and adjust the first chunk's endpoint.
57                           */
58                          range_alloc(asma, range, range->purged,
59                                      pgend + 1, range->pgend);
60                          range_shrink(range, range->pgstart, pgstart - 1);
61                          break;
62                  }
63          }
64
65          return ret;
66  }
```

一块内存默认是处于锁定状态的，当它被解锁之后，就会进入到其宿主匿名共享内存的解锁内存块列表unpinned_list中。只有位于这个列表中的内存块才可以执行锁定操作，因此，第12行到第63行的for循环会在目标匿名共享内存asma的解锁内存块列表unpinned_list中遍历。如果发现一个处于解锁状态的内存块range与指定要锁定的内存块[pgstart, pgend]互相包含或者相交，那么就将它从列表中删除，即重新对它进行锁定，不允许内存管理系统回收。

当一个处于解锁状态的内存块range与指定要锁定的内存块[pgstart, pgend]互相包含或者相交时，第32行的if语句就为true，这时候这两个内存块的位置关系就如图6-2中的A、B、C和D示意图所示。因此，函数ashmem_pin就分四种情况来重新锁定内存块range。

在第一种情况中，指定要锁定的内存块[pgstart, pgend]包含了处于解锁状态的内存块range，即第36行的if语句为true。这时候只要将处于解锁状态的内存块range从其宿主匿名共享内存的解锁内存块列表unpinned_list中删除即可，如第37行代码所示。

在第二种情况中，指定要锁定的内存块[pgstart, pgend]的后半部分与处于解锁状态的内存块range的前半部分相并，即第42行的if语句为true。这时候只需要将处于解锁状态的内存块range的开始地址修改为指定要锁定的内存块的末尾地址的下一个页面地址即可，如第43行代码所示。

在第三种情况中，指定要锁定的内存块[pgstart, pgend]的前半部分与处于解锁状态的内存块range的后半部分相并，即第48行的if语句为true。这时候只需要将处于解锁状态的内存块range的末尾地址修改为指定要锁定的内存块的开始地址的上一个页面地址即可，如第49行代码所示。

在第四种情况中，指定要锁定的内存块[pgstart, pgend]包含在处于解锁状态的内存块range中。这时候相当于指定要锁定的内存块[pgstart, pgend]在处于解锁状态的内存块range的中间挖了一个洞，因此，第58行到第60行代码就将处于解锁状态的内存块range一分为二，划分后得到的两个处于解锁状态的内存块的地址范围分别是[pgend + 1, range->pgend]和[range->pgstart, pgstart - 1]。第58行和第59行调用函数range_alloc为内存块[pgend + 1, range->pgend]分配一个ashmem_range结构体，并且将它加入到其宿主匿名共享内存asma的解锁内存块列表unpinned_list中。第60行调用函数range_shrink将原来就已经处于解锁状态的内存块range的末尾地址修改为pgstart - 1，这样就可以用来描述第二个内存块[range->pgstart, pgstart - 1]了。

函数range_shrink用于修改一个处于解锁状态的内存块的开始地址和末尾地址，它的实现如下所示。

kernel/goldfish/mm/ashmem.c

```
01 /*
02  * range_shrink - shrinks a range
03  *
04  * Caller must hold ashmem_mutex.
05  */
06 static inline void range_shrink(struct ashmem_range *range,
07                                 size_t start, size_t end)
08 {
09         size_t pre = range_size(range);
10
11         range->pgstart = start;
12         range->pgend = end;
13
14         if (range_on_lru(range))
15                 lru_count -= pre - range_size(range);
16 }
```

首先第9行计算处于解锁状态的内存块range的大小，接着第11行和第12行将它的开始地址和末尾地址修改为start和end，最后第14行判断它是否已经被内存管理系统回收。如果没有被回收，那么第15行就调整全局列表ashmem_lru_list中的内存块的总大小，即修改全局变量lru_count的值。

至此，我们就分析完匿名共享内存块的锁定和解锁操作了，其中，处于解锁状态的内存块是可以被内存管理系统回收的。接下来，我们继续分析匿名共享内存块被内存管理系统回收的过程。

6.1.6 匿名共享内存块的回收过程

Ashmem驱动程序在启动时，向内存管理系统注册了一个内存回收函数ashmem_shrink。当系统内存不足时，函数ashmem_shrink就会被调用来回收那些处于解锁状态的匿名共享内存。

函数ashmem_shrink的实现如下所示。

kernel/goldfish/mm/ashmem.c

```
01 /*
02  * ashmem_shrink - our cache shrinker, called from mm/vmscan.c :: shrink_slab
03  *
04  * 'nr_to_scan' is the number of objects (pages) to prune, or 0 to query how
05  * many objects (pages) we have in total.
06  *
07  * 'gfp_mask' is the mask of the allocation that got us into this mess.
08  *
09  * Return value is the number of objects (pages) remaining, or -1 if we cannot
10  * proceed without risk of deadlock (due to gfp_mask).
11  *
12  * We approximate LRU via least-recently-unpinned, jettisoning unpinned partial
13  * chunks of ashmem regions LRU-wise one-at-a-time until we hit 'nr_to_scan'
14  * pages freed.
15  */
16 static int ashmem_shrink(int nr_to_scan, gfp_t gfp_mask)
17 {
18         struct ashmem_range *range, *next;
19
20         /* We might recurse into filesystem code, so bail out if necessary */
21         if (nr_to_scan && !(gfp_mask & __GFP_FS))
22                 return -1;
23         if (!nr_to_scan)
24                 return lru_count;
25
26         mutex_lock(&ashmem_mutex);
27         list_for_each_entry_safe(range, next, &ashmem_lru_list, lru) {
```

```
28                  struct inode *inode = range->asma->file->f_dentry->d_inode;
29                  loff_t start = range->pgstart * PAGE_SIZE;
30                  loff_t end = (range->pgend + 1) * PAGE_SIZE - 1;
31
32                  vmtruncate_range(inode, start, end);
33                  range->purged = ASHMEM_WAS_PURGED;
34                  lru_del(range);
35
36                  nr_to_scan -= range_size(range);
37                  if (nr_to_scan <= 0)
38                          break;
39          }
40          mutex_unlock(&ashmem_mutex);
41
42          return lru_count;
43  }
```

参数nr_to_scan表示内存管理系统希望回收的内存页数。如果它的值等于0，即第23行的if语句为true，那么就表示内存管理系统并不是通知函数ashmem_shrink去执行回收匿名共享内存的操作，而是查询Ashmem驱动程序目前有多少匿名共享内存页可以被回收。因此，第24行将全局变量lru_count的值返回给调用者。

如果参数nr_to_scan的值大于0，那么接下来的第27行到第39行的for循环就遍历全局列表ashmem_lru_list中的解锁内存块，并且逐一地调用函数vmtruncate_range来回收它们所占用的物理页面，直到回收的物理页面数达到nr_to_scan，或者全局列表ashmem_lru_list中已经没有内存可回收为止。

注意

当一块处于解锁状态的内存块range被回收之后，第34行就会将它从全局列表ashmem_lru_list中删除，并且第33行将它的成员变量purged的值设置为ASHMEM_WAS_PURGED，表示它已经被内存管理系统回收了。

函数ashmem_shrink在回收匿名共享内存之前，会先判断参数gfp_mask的__GFP_FS位是否等于0。如果等于0，就表示不允许函数ashmem_shrink执行与文件系统相关的操作。在执行与文件系统相关的操作时，会进一步消耗系统内存，因此，需要禁止它们在内存回收的过程中执行。由于匿名共享内存是在临时文件系统tmpfs的基础上实现的，因此，第21行的if语句如果发现参数gfp_mask的__GFP_FS位等于0，那么第22行就直接返回了。

至此，Ashmem驱动程序的实现就分析完了。接下来，我们分析它在运行时库cutils中的C访问接口。

6.2　运行时库cutils的匿名共享内存访问接口

运行时库cutils的匿名共享内存访问接口实现在ashmem-dev.c文件中，它的目录结构如下：

```
~/Android/system/core
----libcutils
     ----ashmem-dev.c
```

这个文件提供了五个C接口来访问Ashmem驱动程序，它们分别是ashmem_create_region、ashmem_pin_region、ashmem_unpin_region、ashmem_set_prot_region和ashmem_get_size_region。下面我们就分别分析这五个函数的实现。

ashmem_create_region

system/core/libcutils/ashmem-dev.c

```
01 /*
02  * ashmem_create_region - creates a new ashmem region and returns the file
03  * descriptor, or <0 on error
04  *
05  * 'name' is an optional label to give the region (visible in /proc/pid/maps)
06  * 'size' is the size of the region, in page-aligned bytes
07  */
08 int ashmem_create_region(const char *name, size_t size)
09 {
10         int fd, ret;
11
12         fd = open(ASHMEM_DEVICE, O_RDWR);
13         if (fd < 0)
14                 return fd;
15
16         if (name) {
17                 char buf[ASHMEM_NAME_LEN];
18
19                 strlcpy(buf, name, sizeof(buf));
20                 ret = ioctl(fd, ASHMEM_SET_NAME, buf);
21                 if (ret < 0)
22                         goto error;
23         }
24
25         ret = ioctl(fd, ASHMEM_SET_SIZE, size);
26         if (ret < 0)
27                 goto error;
28
29         return fd;
30
31 error:
32         close(fd);
33         return ret;
34 }
```

函数ashmem_create_region用来请求Ashmem驱动程序为应用程序创建一块匿名共享内存，其中，参数name和size分别表示请求创建的匿名共享内存的名称和大小。

请求Ashmem驱动程序创建一块匿名共享内存分三步来完成。

第一步是调用函数open打开设备文件ASHMEM_DEVICE，即设备文件/dev/ashmem，它的定义如下所示。

system/core/libcutils/ashmem-dev.c

```
1 #define ASHMEM_DEVICE   "/dev/ashmem"
```

调用函数open打开设备文件/dev/ashmem时，Ashmem驱动程序的函数ashmem_open就会被调用，主要是为应用程序创建一个ashmem_area结构体，用来描述一块匿名共享内存。

打开了设备文件/dev/ashmem之后，就会得到一个文件描述符，接下来就可以通过这个文件描述符来访问前面请求Ashmem驱动程序创建的匿名共享内存。

第二步是使用 IO控制命令ASHMEM_SET_NAME来请求Ashmem驱动程序将前面所创建的匿名共享内存的名称修改为name。

注意

参数name是可选的。如果它的值等于NULL，那么Ashmem驱动程序就使用默认名称“/dev/ashmem”来命名新创建的匿名共享内存。

第三步是使用IO控制命令ASHMEM_SET_SIZE来请求Ashmem驱动程序将前面所创建的匿名共享内存的大小修改为size。

> **注意**
>
> Ashmem驱动程序在创建一块匿名共享内存时，将它的大小设置为0。

ashmem_pin_region

system/core/libcutils/ashmem-dev.c

```
1 int ashmem_pin_region(int fd, size_t offset, size_t len)
2 {
3         struct ashmem_pin pin = { offset, len };
4         return ioctl(fd, ASHMEM_PIN, &pin);
5 }
```

函数ashmem_pin_region使用IO控制命令ASHMEM_PIN来请求Ashmem驱动程序锁定一小块匿名共享内存。其中，参数fd是前面打开设备文件/dev/ashmem所得到的一个文件描述符；参数offset和len用来指定要锁定的内存块在其宿主匿名共享内存中的偏移地址和长度。

ashmem_unpin_region

system/core/libcutils/ashmem-dev.c

```
1 int ashmem_unpin_region(int fd, size_t offset, size_t len)
2 {
3         struct ashmem_pin pin = { offset, len };
4         return ioctl(fd, ASHMEM_UNPIN, &pin);
5 }
```

函数ashmem_unpin_region使用IO控制命令ASHMEM_UNPIN来请求Ashmem驱动程序解锁一小块匿名共享内存。其中，参数fd是前面打开设备文件/dev/ashmem所得到的一个文件描述符；参数offset和len用来指定要解锁的内存块在其宿主匿名共享内存中的偏移地址和长度。

ashmem_set_prot_region

system/core/libcutils/ashmem-dev.c

```
1 int ashmem_set_prot_region(int fd, int prot)
2 {
3         return ioctl(fd, ASHMEM_SET_PROT_MASK, prot);
4 }
```

函数ashmem_set_prot_region使用IO控制命令ASHMEM_SET_PROT_MASK来请求Ashmem驱动程序修改一块匿名共享内存的访问保护位。其中，参数fd是前面打开设备文件/dev/ashmem所得到的一个文件描述符；参数prot指定要修改的访问保护位，它的取值为PROT_EXEC、PROT_READ、PROT_WRITE或其组合值。

Ashmem驱动程序中的函数set_prot_mask负责处理IO控制命令ASHMEM_SET_PROT_MASK，它的实现如下所示。

kernel/goldfish/mm/ashmem.c

```
01 static int set_prot_mask(struct ashmem_area *asma, unsigned long prot)
02 {
03         int ret = 0;
04
05         mutex_lock(&ashmem_mutex);
06
07         /* the user can only remove, not add, protection bits */
08         if (unlikely((asma->prot_mask & prot) != prot)) {
09                 ret = -EINVAL;
```

```
10                  goto out;
11          }
12
13          /* does the application expect PROT_READ to imply PROT_EXEC? */
14          if ((prot & PROT_READ) && (current->personality & READ_IMPLIES_EXEC))
15                  prot |= PROT_EXEC;
16
17          asma->prot_mask = prot;
18
19 out:
20          mutex_unlock(&ashmem_mutex);
21          return ret;
22 }
```

Ashmem驱动程序在创建一块匿名共享内存时，将它的访问保护位设置为PROT_MASK，表示这块匿名共享内存具有可执行、读和写权限。此后，应用程序只能删除它的访问保护位，而不能增加它的访问保护位。因此，第8行的if语句首先检查要修改的访问保护位prot是否超出了目标匿名共享内存asma所允许的范围。

有一种特殊情况，即当当前进程current的personality属性的READ_IMPLIES_EXEC位被设置为1时，第14行就会检查参数prot的PROT_READ位是否被设置为1。如果是，那么第15行就将它的PROT_EXEC位也设置为1；因为当一个进程的personality属性的READ_IMPLIES_EXEC位被设置为1时，就表示当它有权限读一块内存时，也隐含着它对该内存有执行权限。

最后，第17行将目标匿名共享内存asma的访问保护位prot_mask 设置为参数prot的值。

ashmem_get_size_region

system/core/libcutils/ashmem-dev.c

```
1 int ashmem_get_size_region(int fd)
2 {
3   return ioctl(fd, ASHMEM_GET_SIZE, NULL);
4 }
```

函数ashmem_get_size_region使用IO控制命令ASHMEM_GET_SIZE来请求Ashmem驱动程序返回一块匿名共享内存的大小。其中，参数fd是前面打开设备文件/dev/ashmem所得到的一个文件描述符。

Ashmem驱动程序中的函数ashmem_ioctl负责处理IO控制命令ASHMEM_GET_SIZE，它的实现如下所示。

kernel/goldfish/mm/ashmem.c

```
01 static long ashmem_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
02 {
03          struct ashmem_area *asma = file->private_data;
04          long ret = -ENOTTY;
05
06          switch (cmd) {
07          ......
08          case ASHMEM_GET_SIZE:
09                  ret = asma->size;
10                  break;
11          ......
12          }
13
14          return ret;
15 }
```

第3行首先找到要获得其大小的匿名共享内存asma，接着第9行再将它的大小size返回给调用者。

至此，我们就分析完运行时库cutils中的匿名共享内存的C访问接口了。接下来，我们继续分析在应用程序框架层中提供的匿名共享内存的C++和Java访问接口。

6.3　匿名共享内存的C++访问接口

为了方便应用程序使用匿名共享内存，Android系统在应用程序框架层中提供了两个C++类MemoryHeapBase和MemoryBase来创建和管理匿名共享内存。如果一个进程需要与其他进程共享一块完整的匿名共享内存，那么它就可以使用MemoryHeapBase类创建这块匿名共享内存。如果一个进程创建了一块匿名共享内存之后，只希望与其他进程共享其中的一部分，那么它就可以使用MemoryBase类来创建这块匿名共享内存。

MemoryBase类是在MemoryHeapBase类的基础上实现的，将它们实例化之后就得到一个Service组件，从而可以用来执行进程间通信，即将一块匿名共享内存从一个进程传输到另一个进程。接下来，我们首先分析MemoryHeapBase类的实现，然后再分析MemoryBase类的实现，最后通过一个应用实例来说明它们的使用方法。

6.3.1　MemoryHeapBase

前面提到，MemoryHeapBase类的实例是一个Service组件，相应的，它会有一个对应的Client组件。我们知道，Service组件必须要实现BnInterface接口，并且是运行在Server进程中的；而Client组件必须要实现BpInterface接口，并且是运行在Client进程中的。接下来，我们首先分析MemoryHeapBase类在Server端的实现，然后再分析它在Client端的代理对象的实现。

6.3.1.1　Server端实现

在前面的5.2节的图5-15中，我们描述了Service组件的实现原理，现在将图中的IXXXXXX接口以及BnXXXXXX和XXXXXX类直接替换为IMemoryHeap接口以及BnMemoryHeap和MemoryHeapBase类，就可以得到MemoryHeapBase类在Server端的实现关系图，如图6-3所示。

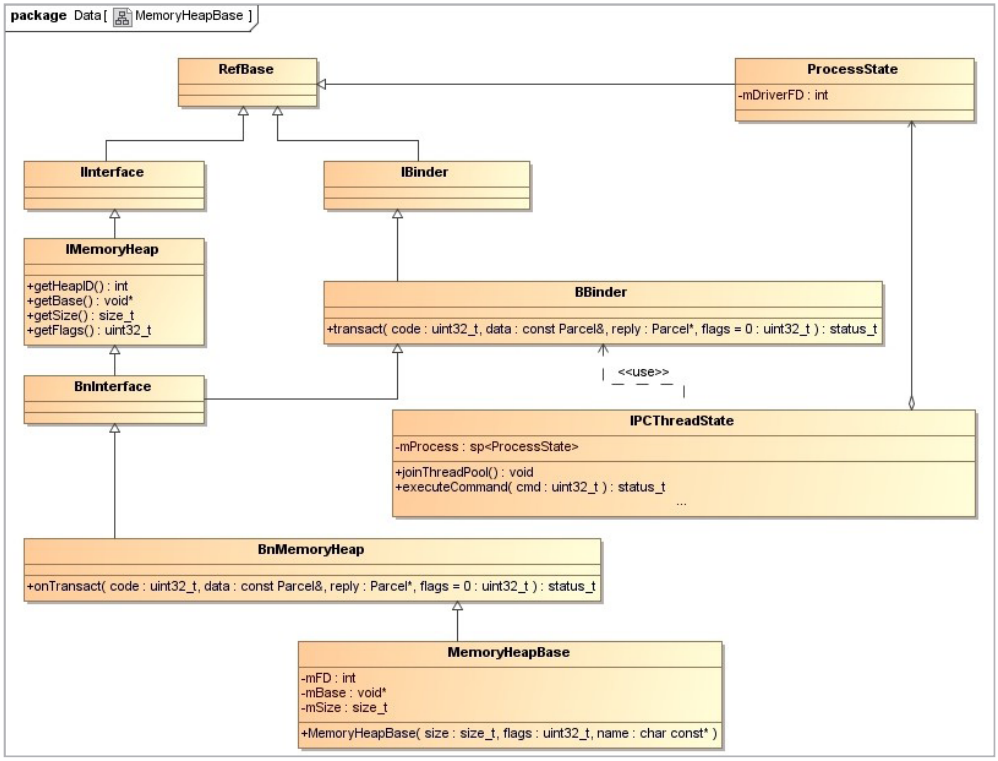


图6-3　MemoryHeapBase类在Server端的实现关系图

图6-3中的类按照其功能不同可以划分为三种类型。第一种类型是和业务相关的，即与匿名共享内存的创建和管理相关的，它们包括三个类IMemoryHeap、BnMemoryHeap和MemoryHeapBase；第二种类型是和Binder进程间通信机制相关的，它们包括六个类IInterface、IBinder、BBinder、BnInterface、ProcessState和IPCThreadState；第三种类型是和智能指针相关的，即RefBase类。在前面的第3章和第5章中，我们已经分析过后面两种类型的类的实现了；因此，在本节中，我们只关注与匿名共享内存创建和管理相关的三个类IMemoryHeap、BnMemoryHeap和MemoryHeapBase的实现。

IMemoryHeap类定义了MemoryHeapBase服务接口，它包含了四个成员函数getHeapID、getBase、getSize和getFlags，分别用来获取一个匿名共享内存块的文件描述符、映射地址、大小和访问保护位，它们的定义如下所示。

frameworks/base/include/binder/IMemory.h

```
01 class IMemoryHeap : public IInterface
02 {
03 public:
04     ......
05
06     virtual int         getHeapID() const = 0;
07     virtual void*       getBase() const = 0;
08     virtual size_t      getSize() const = 0;
09     virtual uint32_t    getFlags() const = 0;
10
11     ......
12 };
```

BnMemoryHeap类是一个Binder本地对象类，它只有一个成员函数onTransact，用来处理Client进程发送过来的HEAP_ID请求，如下所示。

frameworks/base/libs/binder/IMemory.cpp

```
01 status_t BnMemoryHeap::onTransact(
02         uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
03 {
04     switch(code) {
05        case HEAP_ID: {
06             CHECK_INTERFACE(IMemoryHeap, data, reply);
07             reply->writeFileDescriptor(getHeapID());
08             reply->writeInt32(getSize());
09             reply->writeInt32(getFlags());
10             return NO_ERROR;
11         } break;
12         default:
13             return BBinder::onTransact(code, data, reply, flags);
14     }
15 }
```

第7行到第9行代码分别调用由其子类重写的成员函数getHeapID、getSize和getFlags来获取一个匿名共享内存块的文件描述符、大小和访问保护位，并且将它们写入到Parcel对象reply中，以便可以将它们返回给Client进程。

MemoryHeapBase类用来描述一个匿名共享内存服务，它继承了BnMemoryHeap类，并且实现了IMemoryHeap接口的四个成员函数getHeapID、getBase、getSize和getFlags，它的定义如下所示。

frameworks/base/include/binder/MemoryHeapBase.h

```
01 class MemoryHeapBase : public virtual BnMemoryHeap
02 {
03 public:
04     ......
05
06     /*
```

```
07      * maps memory from ashmem, with the given name for debugging
08      */
09     MemoryHeapBase(size_t size, uint32_t flags = 0, char const* name = NULL);
10
11     ......
12
13     /* implement IMemoryHeap interface */
14     virtual int         getHeapID() const;
15     virtual void*       getBase() const;
16     virtual size_t      getSize() const;
17     virtual uint32_t    getFlags() const;
18
19 private:
20     ......
21
22     int         mFD;
23     size_t      mSize;
24     void*       mBase;
25     uint32_t    mFlags;
26
27     ......
28 };
```

成员变量mFD是一个文件描述符，它是打开设备文件/dev/ashmem后得到的，用来描述一个匿名共享内存块；成员变量mSize、mBase和mFlags分别用来描述这块匿名共享内存块的大小、映射地址和访问保护位。

匿名共享内存服务MemoryHeapBase的内部包含了一块匿名共享内存，它的创建过程如下所示。

frameworks/base/libs/binder/MemoryHeapBase.cpp

```
01 MemoryHeapBase::MemoryHeapBase(size_t size, uint32_t flags, char const * name)
02     : mFD(-1), mSize(0), mBase(MAP_FAILED), mFlags(flags),
03       ......
04 {
05     const size_t pagesize = getpagesize();
06     size = ((size + pagesize-1) & ~(pagesize-1));
07     int fd = ashmem_create_region(name == NULL ? "MemoryHeapBase" : name, size);
08     ......
09     if (fd >= 0) {
10         if (mapfd(fd, size) == NO_ERROR) {
11             if (flags & READ_ONLY) {
12                 ashmem_set_prot_region(fd, PROT_READ);
13             }
14         }
15     }
16 }
```

参数size、flags和name分别表示要创建的匿名共享内存块的大小、访问保护位和名称，其中，参数name是可选的。

Ashmem驱动程序为应用程序创建的匿名共享内存的大小是以页面为单位的，因此，第5行首先调用函数getpagesize获得一个页面的大小；接着第6行将参数size的大小对齐到页面边界，即如果参数size的大小不是页面大小的倍数，那么就增加它的值，使得它的大小是一个页面大小的整数倍。

第7行调用由运行时库cutils提供的函数ashmem_create_region来创建一块匿名共享内存，它的返回值是一个文件描述符fd。

注意

如果没有指定要创建的匿名共享内存的名称，即参数name的值为NULL，那么第7行就会使用“MemoryHeapBase”作为要创建的匿名共享内存的名称。

接着第10行调用成员函数mapfd将前面所创建的匿名共享内存块映射到进程的地址空间。最后第11行检查所创建的匿名共享内存块是否是只读的，即参数flags的READ_ONLY位是否等于1。如果是，那么第12行就调用由运行时库cutils提供的函数ashmem_set_prot_region来设置前面所创建的匿名共享内存的访问保护位。

在前面的6.2节中，我们已经分析过函数ashmem_create_region和ashmem_set_prot_region的实现了。因此，接下来我们就重点分析MemoryHeapBase类的成员函数mapfd的实现，它的定义如下所示。

frameworks/base/libs/binder/MemoryHeapBase.cpp

```
01 status_t MemoryHeapBase::mapfd(int fd, size_t size, uint32_t offset)
02 {
03     ......
04
05     if ((mFlags & DONT_MAP_LOCALLY) == 0) {
06         void* base = (uint8_t*)mmap(0, size,
07                 PROT_READ|PROT_WRITE, MAP_SHARED, fd, offset);
08         if (base == MAP_FAILED) {
09             ......
10             close(fd);
11             return -errno;
12         }
13         ......
14         mBase = base;
15         mNeedUnmap = true;
16     } else  {
17         mBase = 0; // not MAP_FAILED
18         mNeedUnmap = false;
19     }
20     mFD = fd;
21     mSize = size;
22     return NO_ERROR;
23 }
```

如果MemoryHeapBase服务的成员变量mFlags的DONT_MAP_LOCALLY位等于1，即第5行的if语句为false，就表示不要将它内部的匿名共享内存块映射到进程的地址空间。这时候第17行将这块匿名共享内存映射到进程地址空间的开始地址mBase设置为0，并且第18行将成员变量mNeedUnmap的值设置为false，表示当这个MemoryHeapBase服务被销毁时，不需要撤销它内部的匿名共享内存的内存映射。

如果MemoryHeapBase服务的成员变量mFlags的DONT_MAP_LOCALLY位等于0，即第5行的if语句为true，那么第6行和第7行就调用函数mmap将它内部的匿名共享内存映射到进程的地址空间。接着第14行将获得的映射地址保存在成员变量mBase中，并且第15行将它的成员变量mNeedUnmap的值设置为true，表示当这个MemoryHeapBase服务被销毁时，需要撤销它内部的匿名共享内存的内存映射。

注意

在将这块匿名共享内存映射到进程的地址空间时，指定这块匿名共享内存的访问保护位为PROT_READ|PROT_WRITE，即只可以对它进行读写，同时，指定要映射的虚拟内存地址的内容是可以和其他进程共享的。

MemoryHeapBase服务的另外四个成员函数getHeapID、getBase、getSize和getFlags的实现如下所示。

frameworks/base/libs/binder/MemoryHeapBase.cpp

```
01 int MemoryHeapBase::getHeapID() const {
02     return mFD;
03 }
04
05 void* MemoryHeapBase::getBase() const {
06     return mBase;
```

```
07 }
08
09 size_t MemoryHeapBase::getSize() const {
10     return mSize;
11 }
12
13 uint32_t MemoryHeapBase::getFlags() const {
14     return mFlags;
15 }
```

它们分别将一个MemoryHeapBase服务内部的匿名共享内存块的文件描述符mFD、映射地址mBase、大小mSize和访问保护位mFlags返回给调用者。

至此，MemoryHeapBase类在Server端的实现就分析完成了，它主要是通过调用运行时库cutils提供的匿名共享内存接口在内部创建一块可以与其他进程共享的匿名共享内存。接下来，我们继续分析MemoryHeapBase类在Client端的实现。

6.3.1.2　Client端实现

MemoryHeapBase类在Client端主要是实现一个类型为BpMemoryHeap的Client组件，即一个实现了IMemoryHeap接口的Binder代理对象，通过它可以获得运行在Server端的MemoryHeapBase服务内部的匿名共享内存块的文件描述符、大小和访问保护位。有了这个匿名共享内存块的信息之后，Client端就可以将它映射到Client进程的地址空间，从而可以访问在Server端创建的匿名共享内存块。

根据前面5.2节中的图5-16所描述的Client组件实现原理图，我们将图中的IXXXXXX接口和BpXXXXXX类直接替换为IMemoryHeap接口和BpMemoryHeap类，就可以得到MemoryHeapBase类在Client端的实现关系图，如图6-4所示。

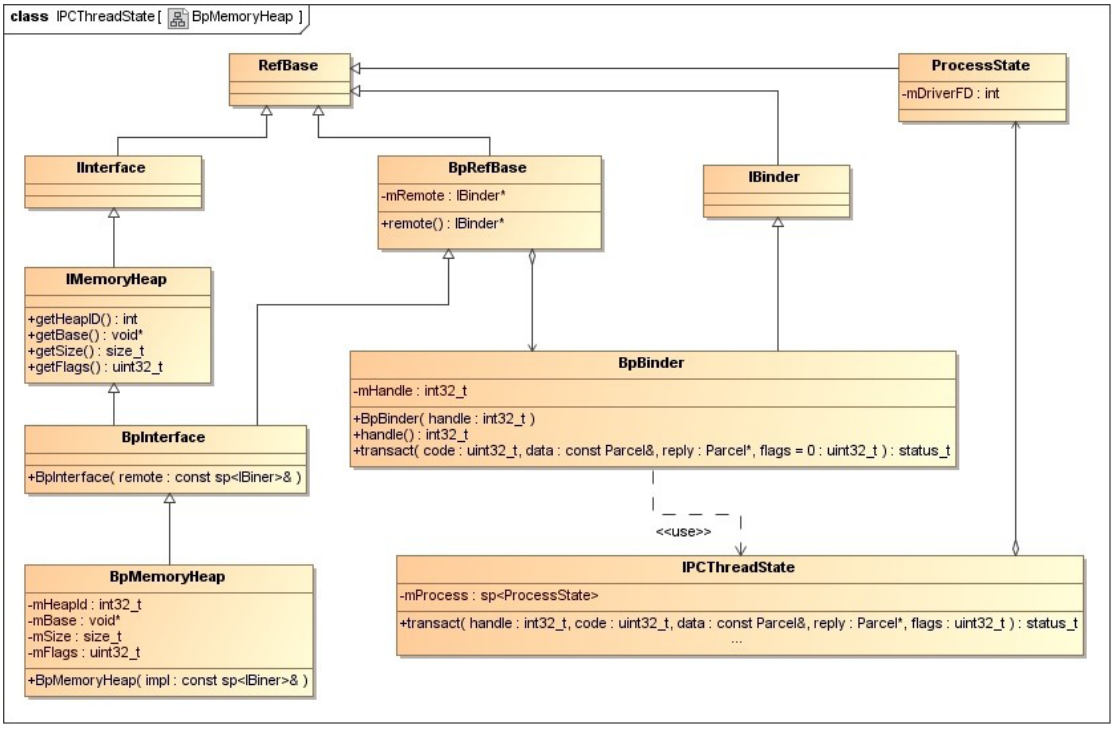


图6-4　MemoryHeapBase类在Client端的实现关系图

图6-4中的类按照其功能不同可以划分为三种类型。第一种类型是和业务相关的，即与匿名共享内存的创建和管理相关的，它们包括两个类IMemoryHeap和BpMemoryHeap；第二种类型是和Binder进程间通信机制相关的，它们包括七个类IInterface、IBinder、BpRefBase、BpBinder、BpInterface、

ProcessState和IPCThreadState；第三种类型是和智能指针相关的，即RefBase类。在前面的第3章、第5章以及6.3.1.1节中，我们已经分析过后面两种类型的类以及IMemoryHeap类的实现了；因此，在本节中，我们只关注BpMemoryHeap类的实现。

BpMemoryHeap类用来描述一个MemoryHeapBase服务的代理对象，它的定义如下所示。

frameworks/base/libs/binder/IMemory.cpp

```
01 class BpMemoryHeap : public BpInterface<IMemoryHeap>
02 {
03 public:
04     BpMemoryHeap(const sp<IBinder>& impl);
05     ......
06
07     virtual int getHeapID() const;
08     virtual void* getBase() const;
09     virtual size_t getSize() const;
10     virtual uint32_t getFlags() const;
11
12     ......
13
14     mutable volatile int32_t mHeapId;
15     mutable void*       mBase;
16     mutable size_t      mSize;
17     mutable uint32_t    mFlags;
18     ......
19 };
```

成员变量mHeapId是一个文件描述符，它是通过请求MemoryHeapBase服务获得的，用来描述一个匿名共享内存块；成员变量mBase、mSize和mFlags分别用来描述这块匿名共享内存的地址、大小和访问保护位。

BpMemoryHeap类所描述的MemoryHeapBase代理对象的创建过程如下所示：

frameworks/base/libs/binder/IMemory.cpp

```
1 BpMemoryHeap::BpMemoryHeap(const sp<IBinder>& impl)
2     : BpInterface<IMemoryHeap>(impl),
3         mHeapId(-1), mBase(MAP_FAILED), mSize(0), mFlags(0), ......
4 {
5 }
```

参数impl指向一个Binder代理对象，它引用了运行在Server端的一个MemoryHeapBase服务，第2行使用它初始化父类BpInterface。

第3行分别将一个MemoryHeapBase代理对象的成员变量mHeapId、mBase、mSize和mFlags的值设置为-1、MAP_FAILED、0和0，表示该MemoryHeapBase代理对象内部的匿名共享内存尚未映射到Client进程的地址空间。

当Client进程第一次访问这个MemoryHeapBase代理对象内部的匿名共享内存块时，它就会请求运行在Server端的MemoryHeapBase服务返回其内部的匿名共享内存块的信息，然后根据返回的信息将这块匿名共享内存映射到Client进程的地址空间，最后就可以访问这块匿名共享内存了。

MemoryHeapBase代理对象的另外四个成员函数getHeapID、getBase、getSize和getFlags的实现如下所示。

frameworks/base/libs/binder/IMemory.cpp

```
01 int BpMemoryHeap::getHeapID() const {
02     assertMapped();
03     return mHeapId;
04 }
05
06 void* BpMemoryHeap::getBase() const {
```

```
07     assertMapped();
08     return mBase;
09 }
10
11 size_t BpMemoryHeap::getSize() const {
12     assertMapped();
13     return mSize;
14 }
15
16 uint32_t BpMemoryHeap::getFlags() const {
17     assertMapped();
18     return mFlags;
19 }
```

它们分别将一个MemoryHeapBase代理对象内部的匿名共享内存块的文件描述符mHeapId、地址mBase、大小mSize和访问保护位mFlags返回给调用者。

当这四个成员函数的其中一个第一次被调用时，成员函数assertMapped就会被调用来请求运行在Server端的MemoryHeapBase服务，返回其内部的匿名共享内存块的文件描述符、大小和访问保护位，以便可以将这块匿名共享内存映射到Client进程的地址空间。

MemoryHeapBase代理对象的成员函数assertMapped的实现如下所示。

frameworks/base/libs/binder/IMemory.cpp

```
01 void BpMemoryHeap::assertMapped() const
02 {
03     if (mHeapId == -1) {
04         sp<IBinder> binder(const_cast<BpMemoryHeap*>(this)->asBinder());
05         sp<BpMemoryHeap> heap(static_cast<BpMemoryHeap*>(find_heap(binder).get()));
06         heap->assertReallyMapped();
07         if (heap->mBase != MAP_FAILED) {
08             Mutex::Autolock _l(mLock);
09             if (mHeapId == -1) {
10                 mBase   = heap->mBase;
11                 mSize   = heap->mSize;
12                 android_atomic_write( dup( heap->mHeapId ), &mHeapId );
13             }
14         } else {
15             // something went wrong
16             free_heap(binder);
17         }
18     }
19 }
```

第3行判断一个MemoryHeapBase代理对象的成员变量mHeapId的值是否等于-1。如果等于-1，那么就说明它尚未将运行在Server端的MemoryHeapBase服务内部的匿名共享内存块映射到Client进程的地址空间来。因此，第4行到第6行代码就会先请求运行在Server端的MemoryHeapBase服务返回其内部的匿名共享内存块的信息，然后根据这些信息将这块匿名共享内存映射到Client进程的地址空间。

Binder库为Client进程维护了一个类型为HeapCache的强指针gHeapCache，它的定义如下所示。

frameworks/base/libs/binder/IMemory.cpp

```
1 static sp<HeapCache> gHeapCache = new HeapCache();
```

在Client进程中，可能会存在多个不同的MemoryHeapBase代理对象引用了同一个MemoryHeapBase服务的情况，在这种情况下，它们内部都引用了同一个Binder代理对象。当其中一个MemoryHeapBase代理对象请求运行在Server端的MemoryHeapBase服务返回其内部的匿名共享内存块的信息之后，它就可以将这块匿名共享内存映射到Client进程的地址空间了。这时候，其他的MemoryHeapBase代理对象就没有必要再将这块匿名共享内存映射到Client进程的地址空间了。

HeapCache类的定义如下所示。

frameworks/base/libs/binder/IMemory.cpp

```
01 class HeapCache : public IBinder::DeathRecipient
02 {
03 public:
04     ......
05
06     sp<IMemoryHeap> find_heap(const sp<IBinder>& binder);
07     ......
08     sp<IMemoryHeap> get_heap(const sp<IBinder>& binder);
09     ......
10
11 private:
12     // For IMemory.cpp
13     struct heap_info_t {
14         sp<IMemoryHeap> heap;
15         int32_t         count;
16     };
17
18     ......
19
20     KeyedVector< wp<IBinder>, heap_info_t > mHeapCache;
21 };
```

HeapCache类内部有一个成员变量mHeapCache，用来描述一个类型为KeyedVector的列表。列表mHeapCache中保存的是Client进程的MemoryHeapBase代理对象，这些MemoryHeapBase代理对象分别使用一个heap_info_t结构体来描述，并且以一个指向它们内部的Binder代理对象的弱指针作为关键字保存在列表mHeapCache中。

由于在Client进程中，引用了同一个MemoryHeapBase服务的不同MemoryHeapBase代理对象都引用了同一个Binder代理对象；因此，当第一个MemoryHeapBase代理对象请求运行在Server端的MemoryHeapBase服务返回其内部的匿名共享内存块的信息时，它就会将自己保存在全局变量gHeapCache内部的列表mHeapCache中。这样，其他的MemoryHeapBase代理对象就可以通过内部的同一个Binder代理对象在全局变量gHeapCache内部的列表mHeapCache中找到第一个MemoryHeapBase代理对象，从而可以获得已经映射到Client进程中的匿名共享内存块的信息。

回到MemoryHeapBase代理对象的成员函数assertMapped中，第4行首先获得一个MemoryHeapBase代理对象内部的Binder代理对象binder，接着第5行将这个Binder代理对象作为参数来调用成员函数find_heap，试图在全局变量gHeapCache中找到一个MemoryHeapBase代理对象heap。找到的MemoryHeapBase代理对象heap就是第一个请求运行在Server端的MemoryHeapBase服务返回其内部的匿名共享内存块的信息的MemoryHeapBase代理对象。

MemoryHeapBase代理对象的成员函数find_heap的实现如下所示。

frameworks/base/libs/binder/IMemory.cpp

```
01 class BpMemoryHeap : public BpInterface<IMemoryHeap>
02 {
03     ......
04
05     static inline sp<IMemoryHeap> find_heap(const sp<IBinder>& binder) {
06         return gHeapCache->find_heap(binder);
07     }
08
09     ......
10 };
```

它调用了全局变量gHeapCache的成员函数find_heap找到一个与Binder代理对象binder对应的MemoryHeapBase代理对象。

HeapCache类的成员函数find_heap的实现如下所示。

frameworks/base/libs/binder/IMemory.cpp

```
01 sp<IMemoryHeap> HeapCache::find_heap(const sp<IBinder>& binder)
02 {
03     Mutex::Autolock _l(mHeapCacheLock);
04     ssize_t i = mHeapCache.indexOfKey(binder);
05     if (i>=0) {
06         heap_info_t& info = mHeapCache.editValueAt(i);
07         ......
08         android_atomic_inc(&info.count);
09         return info.heap;
10     } else {
11         heap_info_t info;
12         info.heap = interface_cast<IMemoryHeap>(binder);
13         info.count = 1;
14         ......
15         mHeapCache.add(binder, info);
16         return info.heap;
17     }
18 }
```

第4行以Binder代理对象binder作为关键字，试图在列表mHeapCache中找到一个对应的heap_info_t结构体。

如果能成功地找到一个对应的heap_info_t结构体，即第5行的if语句为true，那么第9行就将这个heap_info_t结构体的成员变量heap所指向的一个MemoryHeapBase代理对象返回给调用者。在返回之前，第8行会增加这个heap_info_t结构体的成员变量count的值，表示与这个heap_info_t结构体对应的MemoryHeapBase代理对象又多了一个引用者。

如果不能找到一个对应的heap_info_t结构体，即第5行的if语句为false，那么第11行到第13行代码就会先创建一个heap_info_t结构体info，并且将它的成员变量heap指向在内部引用了Binder代理对象binder的MemoryHeapBase代理对象。接着第15行代码就以Binder代理对象binder作为关键字，将heap_info_t结构体info保存在列表mHeapCache中。最后第16行将heap_info_t结构体info的成员变量heap所指向的MemoryHeapBase代理对象返回给调用者。

回到MemoryHeapBase代理对象的成员函数assertMapped中，第6行调用前面所获得的MemoryHeapBase代理对象heap的成员函数assertReallyMapped，来请求运行在Server端的MemoryHeapBase服务返回其内部的匿名共享内存块的信息，并且将该匿名共享内存块映射到Client进程的地址空间；最后就得到该匿名共享内存块在Client进程的映射地址、大小和文件描述符了，分别保存在MemoryHeapBase代理对象heap的成员变量mBase、mSize和mHeapId中。

注意

第10行到第12行代码在将MemoryHeapBase代理对象heap的成员变量mBase、mSize和mHeapId复制到当前MemoryHeapBase代理对象的成员变量mBase、mSize和mHeapId之前，为什么第9行的if语句需要重复判断当前MemoryHeapBase代理对象的成员变量mHeapId是否等于-1呢？前面第3行的if语句不是已经判断过了吗？这是因为当前MemoryHeapBase代理对象与MemoryHeapBase代理对象heap可能是同一个MemoryHeapBase代理对象，因此，为了避免将自己的成员变量mBase、mSize和mHeapId重复复制给自己，第9行就需要再次判断当前MemoryHeapBase代理对象的成员变量mHeapId是否等于-1。

接下来，我们继续分析MemoryHeapBase代理对象的成员函数assertReallyMapped是如何请求运行在Server端的MemoryHeapBase服务返回其内部的匿名共享内存块的信息的。

MemoryHeapBase代理对象的成员函数assertReallyMapped的实现如下所示。

frameworks/base/libs/binder/IMemory.cpp

```
01 void BpMemoryHeap::assertReallyMapped() const
02 {
03     if (mHeapId == -1) {
04         ......
05
06         Parcel data, reply;
07         data.writeInterfaceToken(IMemoryHeap::getInterfaceDescriptor());
08         status_t err = remote()->transact(HEAP_ID, data, &reply);
09         int parcel_fd = reply.readFileDescriptor();
10         ssize_t size = reply.readInt32();
11         uint32_t flags = reply.readInt32();
12
13         ......
14
15         int fd = dup( parcel_fd );
16         ......
17
18         int access = PROT_READ;
19         if (!(flags & READ_ONLY)) {
20             access |= PROT_WRITE;
21         }
22
23         Mutex::Autolock _l(mLock);
24         if (mHeapId == -1) {
25             mRealHeap = true;
26             mBase = mmap(0, size, access, MAP_SHARED, fd, 0);
27             if (mBase == MAP_FAILED) {
28                 ......
29                 close(fd);
30             } else {
31                 mSize = size;
32                 mFlags = flags;
33                 android_atomic_write(fd, &mHeapId);
34             }
35         }
36     }
37 }
```

第3行首先判断成员变量mHeapId的值是否等于-1。如果不等于-1，那么就说明该MemoryHeapBase代理对象已经请求过运行在Server端的MemoryHeapBase服务返回其内部的匿名共享内存块的信息了。因此，这时候函数什么也不用做就可以返回了。

第8行通过成员函数remote来获得该MemoryHeapBase代理对象内部的Binder代理对象，接着再调用这个Binder代理对象的成员函数transact向运行在Server端的MemoryHeapBase服务发送一个类型为HEAP_ID的进程间通信请求，即请求该MemoryHeapBase服务返回其内部的匿名共享内存的文件描述符、大小和访问保护位。

MemoryHeapBase服务将其内部的匿名共享内存的文件描述符经过Binder驱动程序返回给Client进程时，Binder驱动程序会在Client进程中复制这个文件描述符，使得它指向MemoryHeapBase服务中的匿名共享内存。在后面的6.5节中，我们再详细描述这个过程。

获得了一个指向运行在Server端的MemoryHeapBase服务内部的匿名共享内存的文件描述符之后，接着第26行就可以调用函数mmap将这块匿名共享内存映射到Client进程的地址空间，最后就可以获得它的地址，并且保存在成员变量mBase中。这样，Client进程就可以通过这个地址来访问这块匿名共享

内存了，即与Server端共享了同一块匿名共享内存。

注意

第26行调用函数mmap将运行在Server端的MemoryHeapBase服务内部的匿名共享内存映射到Client进程的地址空间之前，为什么第24行的if语句需要重复判断当前MemoryHeapBase代理对象的成员变量mHeapId是否等于-1呢？前面第3行的if语句不是已经判断过了吗？这是因为第8行的代码是一个同步的Binder进程间通信过程，在通信过程中，当前线程可能会进入睡眠状态。在当前线程进入睡眠状态的过程中，可能其他线程正在调用当前BpMemoryHeap对象的成员函数assertReallyMapped来请求运行在Server端的MemoryHeapBase服务返回其内部的匿名共享内存的信息。当这个其他的线程先于当前线程获得这块匿名共享内存的信息时，它就会将这块匿名共享内存映射到Client进程的地址空间。因此，第24行的if语句就需要重复判断当前BpMemoryHeap对象的成员变量mHeapId的值是否等于-1。如果不等于，那么就不需要重复将这块匿名共享内存映射到Client进程的地址空间了。

至此，MemoryHeapBase类在Server端和Client端的实现就分析完成了。接下来，我们继续分析另外一个匿名共享内存的C++访问接口MemoryBase的实现。

6.3.2　MemoryBase

前面提到，MemoryBase类是在MemoryHeapBase类的基础上实现的，它们都可以用来描述一个匿名共享内存服务，它们的关系如图6-5所示。

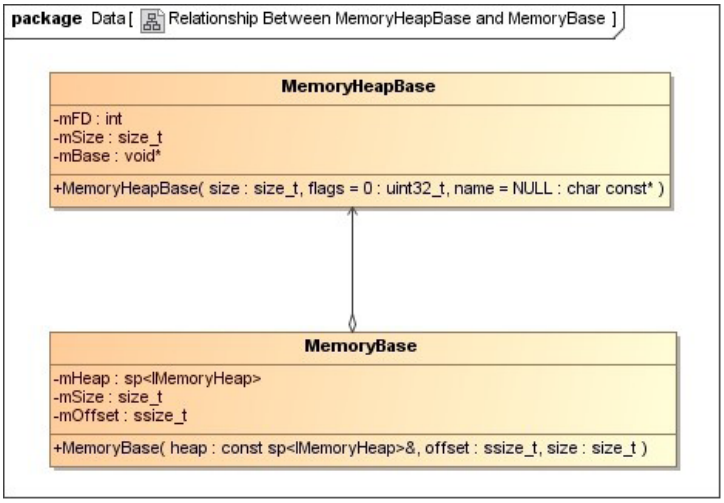


图6-5　MemoryBase类和MemoryHeapBase类的关系图

MemoryBase类内部有一个类型为IMemoryHeap的强指针mHeap，它指向一个MemoryHeapBase服务，MemoryBase类就是通过它来描述一个匿名共享内存服务的。MemoryBase类所维护的匿名共享内存其实只是其内部的MemoryHeapBase服务所维护的匿名共享内存中的一小块。这一小块匿名共享内存用另外两个成员变量mOffset和mSize来描述；其中，mOffset表示这一小块匿名共享内存在一块完整的匿名共享内存中的偏移值，mSize则表示这一小块匿名共享内存的大小。

类似于MemoryHeapBase类，MemoryBase类的实现也分为Server端和Client端两个部分；因此，接下来我们首先分析它在Server端的实现，然后再分析它在Client端的实现。

6.3.2.1 Server端实现

在前面的5.2节的图5-15中，我们描述了Service组件的实现原理，现在将图中的IXXXXXX接口以及BnXXXXXX和XXXXXX类直接替换为IMemory接口以及BnMemory和MemoryBase类，就可以得到MemoryBase类在Server端的实现关系图，如图6-6所示。

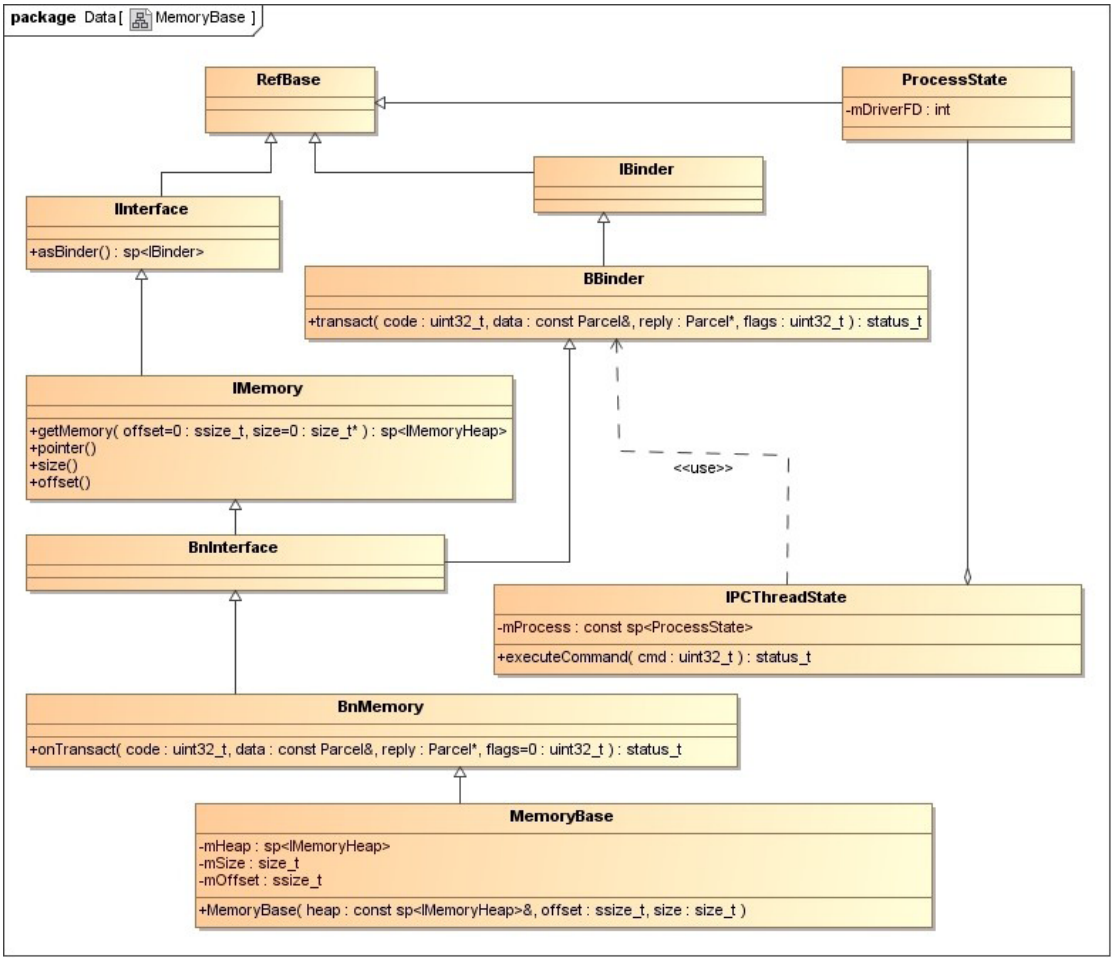


图6-6 MemoryBase类在Server端的实现关系图

MemoryBase类在Server端的实现与MemoryHeapBase类在Server端的实现类似，只要将图6-6中的三个类IMemory、BnMemory和MemoryBase分别替换成IMemoryHeap、BnMemoryHeap和MemoryHeapBase类，就可以得到图6-3所描述的MemoryHeapBase类在Server端的实现关系图。接下来我们主要分析三个类IMemory、BnMemory和MemoryBase的实现。

IMemory类定义了MemoryBase服务接口，它主要包含四个成员函数getMemory、pointer、size和offset，它们的定义如下所示。

frameworks/base/include/binder/IMemory.h

```
01 class IMemory : public IInterface
02 {
03 public:
04     ......
05     virtual sp<IMemoryHeap> getMemory(ssize_t* offset=0, size_t* size=0) const = 0;
06
07     ......
08     void* pointer() const;
```

```
09      size_t size() const;
10      ssize_t offset() const;
11 };
```

成员函数getMemory用来获取一个MemoryBase服务内部的MemoryHeapBase服务。当它被调用时，如果指定的参数offset和size不等于NULL，那么它还会返回这个MemoryBase服务内部的一小块匿名共享内存的偏移值和大小，这两个信息同样可以通过成员函数size和offset来获取。成员函数pointer用来获取一个MemoryBase服务内部的小块匿名共享内存的地址。

在这四个成员函数中，只有getMemory是纯虚函数，需要由其子类来重写，其余三个成员函数是由IMemory类实现的，如下所示。

frameworks/base/libs/binder/IMemory.cpp

```
01 void* IMemory::pointer() const {
02      ssize_t offset;
03      sp<IMemoryHeap> heap = getMemory(&offset);
04      void* const base = heap!=0 ? heap->base() : MAP_FAILED;
05      if (base == MAP_FAILED)
06          return 0;
07      return static_cast<char*>(base) + offset;
08 }
09
10 size_t IMemory::size() const {
11      size_t size;
12      getMemory(NULL, &size);
13      return size;
14 }
15
16 ssize_t IMemory::offset() const {
17      ssize_t offset;
18      getMemory(&offset);
19      return offset;
20 }
```

成员函数pointer首先调用由其子类重写的成员函数getMemory来获取一个MemoryHeapBase服务heap，以及一个偏移值offset，接着再获得这个MemoryHeapBase服务内部的匿名共享内存的地址base；最后将变量base和offset的值相加，就得到了一个MemoryBase服务内部的一小块匿名共享内存的地址。成员函数size和offset也是通过调用由子类重写的成员函数getMemory来获取一个MemoryBase服务内部的一小块匿名共享内存的大小和偏移值的。

BnMemory类是一个Binder本地对象类，它只有一个成员函数onTransact，用来处理Client进程发送过来的GET_MEMORY请求，如下所示。

frameworks/base/libs/binder/IMemory.cpp

```
01 status_t BnMemory::onTransact(
02      uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
03 {
04      switch(code) {
05          case GET_MEMORY: {
06              CHECK_INTERFACE(IMemory, data, reply);
07              ssize_t offset;
08              size_t size;
09              reply->writeStrongBinder( getMemory(&offset, &size)->asBinder() );
10              reply->writeInt32(offset);
11              reply->writeInt32(size);
12              return NO_ERROR;
13          } break;
14          default:
15              return BBinder::onTransact(code, data, reply, flags);
16      }
17 }
```

第9行首先调用由其子类实现的成员函数getMemory来获得一个MemoryHeapBase服务，以及一小块匿名共享内存的偏移值和大小，接着再将它们写入到Parcel对象reply中，以便可以将它们返回给Client进程。

MemoryBase类用来描述一个匿名共享内存服务，它继承了BnMemory类，并且实现了IMemory接口的成员函数getMemory，它的定义如下所示。

frameworks/base/include/binder/MemoryBase.h

```
01 class MemoryBase : public BnMemory
02 {
03 public:
04     MemoryBase(const sp<IMemoryHeap>& heap, ssize_t offset, size_t size);
05     ......
06     virtual sp<IMemoryHeap> getMemory(ssize_t* offset, size_t* size) const;
07
08     ......
09
10 private:
11     size_t          mSize;
12     ssize_t         mOffset;
13     sp<IMemoryHeap> mHeap;
14 };
```

成员变量mHeap指向一个MemoryHeapBase服务；成员变量mSize和mOffset分别用来描述一小块匿名共享内存的大小和偏移值，它们都是在构造函数中初始化的，如下所示。

frameworks/base/libs/binder/MemoryBase.cpp

```
1 MemoryBase::MemoryBase(const sp<IMemoryHeap>& heap,
2         ssize_t offset, size_t size)
3     : mSize(size), mOffset(offset), mHeap(heap)
4 {
5 }
```

MemoryBase类的成员函数getMemory的实现如下所示。

frameworks/base/libs/binder/MemoryBase.cpp

```
1 sp<IMemoryHeap> MemoryBase::getMemory(ssize_t* offset, size_t* size) const
2 {
3     if (offset) *offset = mOffset;
4     if (size)   *size = mSize;
5     return mHeap;
6 }
```

第3行和第4行的if语句分别判断参数offset和size的值是否不等于NULL。如果不等于，那么函数就分别将成员变量mOffset和mSize所描述的一小块匿名共享内存的偏移值和大小设置到它们的地址中。最后第5行将成员变量mHeap所指向的一个MemoryHeapBase服务返回给调用者。

至此，MemoryBase类在Server端的实现就分析完成了，它主要通过一个MemoryHeapBase服务在内部维护一小块可以与其他进程共享的匿名共享内存。接下来，我们继续分析MemoryBase类在Client端的实现。

6.3.2.2 Client端实现

MemoryBase类在Client端主要是实现一个类型为BpMemory的Client组件，即一个实现了IMemory接口的Binder代理对象，通过它可以获得运行在Server端的MemoryBase服务内部的一小块匿名共享内存的偏移值和大小，以及一个MemoryHeapBase代理对象。有了这个MemoryHeapBase代理对象之后，Client端就可以将在Server端创建的一块匿名共享内存映射到自己的地址空间了。

根据前面5.2节中的图5-16所描述的Client组件实现原理图，我们将图中的IXXXXXX接口和

BpXXXXXX类直接替换为IMemory接口和BpMemory类，就可以得到MemoryBase类在Client端的实现关系图，如图6-7所示。

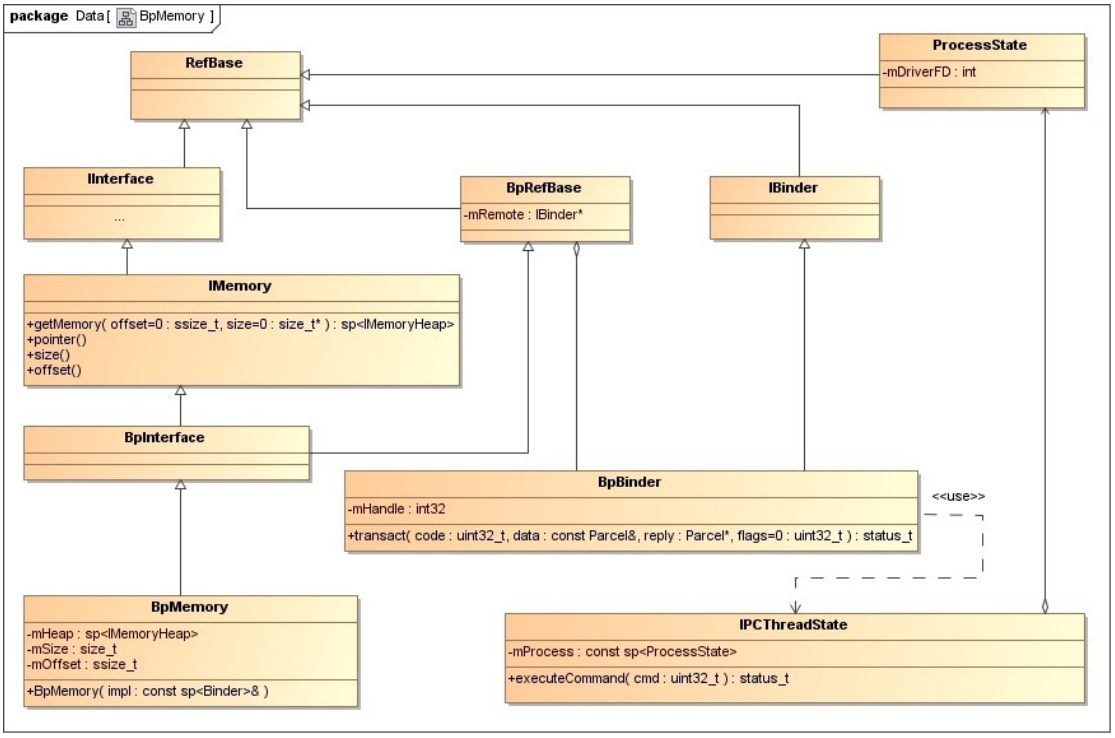


图6-7　MemoryBase类在Client端的实现关系图

MemoryBase类在Client端的实现与MemoryHeapBase类在Client端的实现类似，只要把图6-7中的两个类IMemory和BpMemory分别替换成IMemoryHeap和BpMemoryHeap类，就可以得到图6-4所描述的MemoryHeapBase类在Client端的实现关系图了。在前面的6.3.2.1节中，我们已经分析过IMemory类的实现了，因此，在本节中，我们只关注BpMemory类的实现。

BpMemory类用来描述一个MemoryBase服务的代理对象，它的定义如下所示。

frameworks/base/libs/binder/IMemory.cpp

```
01 class BpMemory : public BpInterface<IMemory>
02 {
03 public:
04     BpMemory(const sp<IBinder>& impl);
05     ......
06     virtual sp<IMemoryHeap> getMemory(ssize_t* offset=0, size_t* size=0) const;
07
08 private:
09     mutable sp<IMemoryHeap> mHeap;
10     mutable ssize_t mOffset;
11     mutable size_t mSize;
12 };
```

成员变量mHeap指向一个MemoryHeapBase代理对象；成员变量mOffset和mSize分别用来描述一小块匿名共享内存的偏移值和大小。

BpMemory类所描述的MemoryBase代理对象的创建过程如下所示。

frameworks/base/libs/binder/IMemory.cpp

```
1 BpMemory::BpMemory(const sp<IBinder>& impl)
2     : BpInterface<IMemory>(impl), mOffset(0), mSize(0)
3 {
4 }
```

参数impl指向一个Binder代理对象，它引用了运行在Server端的一个MemoryBase服务，第2行使用它初始化父类BpInterface。第2行还将一个MemoryBase代理对象的成员变量mOffset和mSize的值设置为0，表示该MemoryBase代理对象内部的一小块匿名共享内存尚未映射到Client进程的地址空间。

当Client进程第一次访问这个MemoryBase代理对象内部的一小块匿名共享内存时，它就会请求运行在Server端的MemoryBase服务返回其内部的一个MemoryHeapBase服务代理对象，然后通过这个MemoryHeapBase代理对象将在Server端创建的一块匿名共享内存映射到Client进程的地址空间。

BpMemory类的成员函数getMemory的实现如下所示。

frameworks/base/libs/binder/IMemory.cpp

```
01 sp<IMemoryHeap> BpMemory::getMemory(ssize_t* offset, size_t* size) const
02 {
03     if (mHeap == 0) {
04         Parcel data, reply;
05         data.writeInterfaceToken(IMemory::getInterfaceDescriptor());
06         if (remote()->transact(GET_MEMORY, data, &reply) == NO_ERROR) {
07             sp<IBinder> heap = reply.readStrongBinder();
08             ssize_t o = reply.readInt32();
09             size_t s = reply.readInt32();
10             if (heap != 0) {
11                 mHeap = interface_cast<IMemoryHeap>(heap);
12                 if (mHeap != 0) {
13                     mOffset = o;
14                     mSize = s;
15                 }
16             }
17         }
18     }
19     if (offset) *offset = mOffset;
20     if (size) *size = mSize;
21     return mHeap;
22 }
```

第3行的if语句判断一个MemoryBase代理对象的成员变量mHeap的值是否等于0。如果不等于0，那么就说明该MemoryBase代理对象已经请求过运行在Server端的MemoryBase服务返回其内部的一个MemoryHeapBase服务代理对象，以及一小块匿名共享内存的信息了。因此，第19行到第21行代码就可以直接将这一小块匿名共享内存的偏移值mOffset和大小mSize，以及这个MemoryHeapBase代理对象mHeap返回给调用者了。

第6行首先通过成员函数remote来获得该MemoryBase代理对象内部的Binder代理对象，然后调用这个Binder代理对象的成员函数transact向运行在Server端的MemoryBase服务发送一个类型为GET_MEMORY的进程间通信请求，即请求该MemoryBase服务返回其内部的一个MemoryHeapBase服务代理对象，以及一小块匿名共享内存的偏移值和大小。接着第8行到第16行代码将这个MemoryHeapBase代理对象，以及这一小块匿名共享内存的偏移值和大小分别保存在该MemoryBase代理对象的成员变量mHeap、mOffset和mSize中。最后第19行到第21行代码将这些信息返回给调用者。

至此，MemoryBase类在Server端和Client端的实现就分析完成了。接下来，我们通过一个应用实例来说明它们的使用方法。

6.3.3 应用实例

在本节中，我们将创建两个应用程序SharedBufferServer和SharedBufferClient。应用程序SharedBufferServer在启动时，会创建一个共享缓冲区服务SharedBufferService，并且将它注册到Service Manager中，以便可以提供一段共享缓冲区给其他应用程序使用。应用程序SharedBufferClient

在启动时，会请求Service Manager返回在应用程序SharedBufferServer中创建的共享缓冲区服务SharedBufferService的一个代理对象，接着就通过这个代理对象将在应用程序SharedBufferServer中创建的一段共享缓冲区映射到本进程的地址空间，最后就可以使用这段共享缓冲区了。共享缓冲区服务SharedBufferService实现了共享缓冲区服务接口ISharedBuffer，它是在匿名共享内存服务MemoryBase的基础上来实现一段共享缓冲区的。

应用程序SharedBufferServer和SharedBufferClient由三个模块组成：第一个是公共模块，用来定义SharedBufferService服务接口ISharedBuffer、Binder本地对象类BnSharedBuffer和Binder代理对象类BpSharedBuffer；第二个是Server端模块，用来实现应用程序SharedBufferServer；第三个是Client端模块，用来实现应用程序SharedBufferClient。

这三个模块的目录结构如下：

```
~/Android/external/ashmem
----common
     ----ISharedBuffer.h
     ----ISharedBuffer.cpp
----server
     ----SharedBufferServer.cpp
     ----Android.mk
----client
     ----SharedBufferClient.cpp
     ----Android.mk
```

common模块由文件ISharedBuffer.h和ISharedBuffer.cpp组成，server模块由文件SharedBufferServer.cpp和Android.mk组成；client模块由文件SharedBufferClient.cpp和Android.mk组成。接下来，我们就分别介绍各个文件的内容。

common/ISharedBuffer.h

```
01 #ifndef ISHAREDBUFFER_H_
02 #define ISHAREDBUFFER_H_
03
04 #include <utils/RefBase.h>
05 #include <binder/IInterface.h>
06 #include <binder/Parcel.h>
07
08 #define SHARED_BUFFER_SERVICE "shy.luo.SharedBuffer"
09 #define SHARED_BUFFER_SIZE 4
10
11 using namespace android;
12
13 class ISharedBuffer: public IInterface
14 {
15 public:
16         DECLARE_META_INTERFACE(SharedBuffer);
17         virtual sp<IMemory> getBuffer() = 0;
18 };
19
20 class BnSharedBuffer: public BnInterface<ISharedBuffer>
21 {
22 public:
23         virtual status_t onTransact(uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags = 0);
24 };
25
26 #endif
```

第13行到第18行定义了共享缓冲区服务接口ISharedBuffer，它只有一个成员函数getBuffer，用来返回一个匿名共享内存服务接口IMemory。

第20行到第24行定义了共享缓冲区服务必须要继承的Binder本地对象类BnSharedBuffer，它只有一个成员函数onTransact，用来接收进程间通信请求。

此外，第8行和第9行定义了两个宏SHARED_BUFFER_SERVICE和SHARED_BUFFER_SIZE，其中，SHARED_BUFFER_SERVICE是一个字符串，用来描述共享缓冲区服务SharedBufferService注册到Service Manager中的名称；而SHARED_BUFFER_SIZE是一个整数，用来描述共享缓冲区服务SharedBufferService内部所创建的共享缓冲区的大小。

common/ISharedBuffer.cpp

```
01 #define LOG_TAG "ISharedBuffer"
02
03 #include <utils/Log.h>
04 #include <binder/MemoryBase.h>
05
06 #include "ISharedBuffer.h"
07
08 using namespace android;
09
10 enum
11 {
12         GET_BUFFER = IBinder::FIRST_CALL_TRANSACTION
13 };
14
15 class BpSharedBuffer: public BpInterface<ISharedBuffer>
16 {
17 public:
18         BpSharedBuffer(const sp<IBinder>& impl)
19                 : BpInterface<ISharedBuffer>(impl)
20         {
21
22         }
23
24 public:
25         sp<IMemory> getBuffer()
26         {
27                 Parcel data;
28                 data.writeInterfaceToken(ISharedBuffer::getInterfaceDescriptor());
29
30                 Parcel reply;
31                 remote()->transact(GET_BUFFER, data, &reply);
32
33                 sp<IMemory> buffer = interface_cast<IMemory>(reply.readStrongBinder());
34
35                 return buffer;
36         }
37 };
38
39 IMPLEMENT_META_INTERFACE(SharedBuffer, "shy.luo.ISharedBuffer");
40
41 status_t BnSharedBuffer::onTransact(uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
42 {
43         switch(code)
44         {
45                 case GET_BUFFER:
46                 {
47                         CHECK_INTERFACE(ISharedBuffer, data, reply);
48
49                         sp<IMemory> buffer = getBuffer();
50                         if(buffer != NULL)
51                         {
```

```
52                                  reply->writeStrongBinder(buffer->asBinder());
53                          }
54
55                          return NO_ERROR;
56                  }
57                  default:
58                  {
59                          return BBinder::onTransact(code, data, reply, flags);
60                  }
61          }
62  }
```

第15行到第37行定义了共享缓冲区服务SharedBufferService的Binder代理对象类BpSharedBuffer，用来描述一个共享缓冲区服务SharedBufferService的代理对象。

BpSharedBuffer类的成员函数getBuffer通过内部的一个Binder代理对象向应用程序SharedBufferServer发送一个类型为GET_BUFFER的进程间通信请求，以便应用程序SharedBufferServer可以向它返回一个MemoryBase服务代理对象。

第41行到第62行定义了共享缓冲区服务SharedBufferService的Binder本地对象类BnSharedBuffer的成员函数onTransact，用来处理从应用程序SharedBufferClient发送过来的类型为GET_BUFFER的进程间通信请求。第49行首先调用由其子类重写的成员函数getBuffer来获得一个MemoryBase服务，接着第52行将这个MemoryBase服务写入到Parcel对象reply中，以便可以返回给应用程序SharedBufferClient。

server/SharedBufferServer.cpp

```
01 #define LOG_TAG "SharedBufferServer"
02
03 #include <utils/Log.h>
04 #include <binder/MemoryBase.h>
05 #include <binder/MemoryHeapBase.h>
06 #include <binder/IServiceManager.h>
07 #include <binder/IPCThreadState.h>
08
09 #include "../common/ISharedBuffer.h"
10
11 class SharedBufferService : public BnSharedBuffer
12 {
13 public:
14          SharedBufferService()
15          {
16                  sp<MemoryHeapBase> heap = new MemoryHeapBase(SHARED_BUFFER_SIZE, 0, "SharedBuffer");
17                  if(heap != NULL)
18                  {
19                          mMemory = new MemoryBase(heap, 0, SHARED_BUFFER_SIZE);
20
21                          int32_t* data = (int32_t*)mMemory->pointer();
22                          if(data != NULL)
23                          {
24                                  *data = 0;
25                          }
26                  }
27          }
28
29          virtual ~SharedBufferService()
30          {
31                  mMemory = NULL;
32          }
33
34 public:
35          static void instantiate()
```

```
36          {
37          defaultServiceManager()->addService(String16(SHARED_BUFFER_SERVICE), new SharedBufferService());
38          }
39
40          virtual sp<IMemory> getBuffer()
41          {
42                  return mMemory;
43          }
44
45  private:
46          sp<MemoryBase> mMemory;
47  };
48
49  int main(int argc, char** argv)
50  {
51          SharedBufferService::instantiate();
52
53          ProcessState::self()->startThreadPool();
54          IPCThreadState::self()->joinThreadPool();
55
56          return 0;
57  }
```

第11行到第47行定义了共享缓冲区服务SharedBufferService类，它继承了Binder本地对象类BnSharedBuffer，并且实现了ISharedBuffer接口的成员函数getBuffer。

在SharedBufferService类的构造函数中，第16行首先创建一个MemoryHeapBase服务heap，接着第19行通过MemoryHeapBase服务heap来创建一个MemoryBase服务，并且保存在成员变量mMemory中，最后第21行到第25行将在MemoryHeapBase服务heap中创建的一块匿名共享内存的内容初始化为0。由于这块匿名共享内存的大小等于4字节，因此，我们可以把它当作一个整型缓冲区来使用。

在SharedBufferService类的静态成员函数instantiate中，第37行首先创建一个SharedBufferService服务，接着再将它注册到Service Manager中。

第49行到第57行定义了应用程序SharedBufferServer的入口函数main。第51行首先调用SharedBufferService类的静态成员函数instantiate创建和注册一个SharedBufferService服务，接着第53行启动一个Binder线程池，最后第54行将当前线程加入到前面所启动的Binder线程池中，以便可以接收和处理进程间通信请求。

server/Android.mk

```
1 LOCAL_PATH := $(call my-dir)
2 include $(CLEAR_VARS)
3 LOCAL_MODULE_TAGS := optional
4 LOCAL_SRC_FILES := ../common/ISharedBuffer.cpp \
5         SharedBufferServer.cpp
6 LOCAL_SHARED_LIBRARIES:= libcutils libutils libbinder
7 LOCAL_MODULE := SharedBufferServer
8 include $(BUILD_EXECUTABLE)
```

这是应用程序SharedBufferServer的编译脚本文件，用来编译源文件ISharedBuffer.cpp和SharedBufferServer.cpp。

client/SharedBufferClient.cpp

```
01 #define LOG_TAG "SharedBufferClient"
02
03 #include <utils/Log.h>
04 #include <binder/MemoryBase.h>
05 #include <binder/IServiceManager.h>
```

```
06
07 #include "../common/ISharedBuffer.h"
08
09 int main()
10 {
11         sp<IBinder> binder = defaultServiceManager()->getService(String16(SHARED_BUFFER_SERVICE));
12         if(binder == NULL)
13         {
14                 printf("Failed to get service: %s.\n", SHARED_BUFFER_SERVICE);
15                 return -1;
16         }
17
18         sp<ISharedBuffer> service = ISharedBuffer::asInterface(binder);
19         if(service == NULL)
20         {
21                 return -2;
22         }
23
24         sp<IMemory> buffer = service->getBuffer();
25         if(buffer == NULL)
26         {
27                 return -3;
28         }
29
30         int32_t* data = (int32_t*)buffer->pointer();
31         if(data == NULL)
32         {
33                 return -4;
34         }
35
36         printf("The value of the shared buffer is %d.\n", *data);
37
38         *data = *data + 1;
39
40         printf("Add value 1 to the shared buffer.\n");
41
42         return 0;
43 }
```

第9行到第43行定义了应用程序SharedBufferClient的入口函数main。第11行首先通过Service Manager来获得运行在应用程序SharedBufferServer中的SharedBufferService服务的一个Binder代理对象binder，接着第18行将Binder代理对象binder封装成一个SharedBufferService代理对象service。有了SharedBufferService代理对象service之后，第24行就可以通过它来获得一个MemoryBase代理对象buffer了。

第30行调用MemoryBase代理对象buffer的成员函数pointer将在应用程序SharedBufferServer中创建的一块匿名共享内存映射到本进程的地址空间，并且获得它的访问地址data。有了这个地址之后，第36行首先将它里面的整数值读取出来，接着第38行将读取到的整数值增加1，并且写入到原来的匿名共享内存中。

当我们重复运行应用程序SharedBufferClient时，如果每次打印出来的整数值都比上一次大1，那么就说明我们成功地在应用程序SharedBufferServer和SharedBufferClient中共享一段缓冲区了，即共享一块匿名共享内存了。

client/Android.mk

```
1 LOCAL_PATH := $(call my-dir)
2 include $(CLEAR_VARS)
3 LOCAL_MODULE_TAGS := optional
4 LOCAL_SRC_FILES := ../common/ISharedBuffer.cpp \
5         SharedBufferClient.cpp
```

```
6 LOCAL_SHARED_LIBRARIES:= libcutils libutils libbinder
7 LOCAL_MODULE := SharedBufferClient
8 include $(BUILD_EXECUTABLE)
```

这是应用程序SharedBufferClient的编译脚本文件，用来编译源文件ISharedBuffer.cpp和SharedBufferClient.cpp。

上述文件都准备好之后，我们就可以对应用程序SharedBufferServer和SharedBufferClient进行编译和打包了。

```
USER@MACHINE:~/Android$ mmm ./external/ashmem/server/
USER@MACHINE:~/Android$ mmm ./external/ashmem/client/
USER@MACHINE:~/Android$ make snod
```

编译成功之后，就可以在out/target/product/gerneric/system/bin目录下看到应用程序文件SharedBufferServer和SharedBufferClient了；打包成功之后，应用程序SharedBufferServer和SharedBufferClient就包含在out/target/product/gerneric目录下的Android系统镜像文件system.img中了。

最后，我们使用新得到的系统镜像文件system.img来启动Android模拟器。Android模拟器启动起来之后，使用adb工具连接上它，并且进入到/system/bin目录中，将应用程序SharedBufferServer和SharedBufferClient运行起来，以便验证它们是否共享了一段缓冲区。

```
USER@MACHINE: ~/Android$ emulator -kernel kernel/goldfish/arch/arm/boot/zImage &
USER@MACHINE: ~/Android$ adb shell
root@android:/ # cd system/bin
root@android:/system/bin # ./SharedBufferServer &
root@android:/system/bin # ./SharedBufferClient
The value of the shared buffer is 0.
Add value 1 to the shared buffer.
root@android:/system/bin # ./SharedBufferClient
The value of the shared buffer is 1.
Add value 1 to the shared buffer.
root@android:/system/bin # ./SharedBufferClient
The value of the shared buffer is 2.
Add value 1 to the shared buffer.
root@android:/system/bin # ./SharedBufferClient
The value of the shared buffer is 3.
Add value 1 to the shared buffer.
```

应用程序SharedBufferServer启动起来之后，再连续四次运行应用程序SharedBufferClient；如果打印出来的整数分别为0、1、2和3，那么就说明应用程序SharedBufferServer和SharedBufferClient成功地共享一段缓冲区了，即成功地共享一块匿名共享内存了。

6.4 匿名共享内存的Java访问接口

为了方便Android应用程序使用匿名共享内存，Android系统在应用程序框架层中提供了Java类MemoryFile来创建和管理匿名共享内存。使用Java类MemoryFile创建的匿名共享内存可以在不同的Android应用程序之间进行共享。在本节中，我们首先分析Java类MemoryFile的实现原理，然后通过一个应用实例来说明它的使用方法。

6.4.1 MemoryFile

MemoryFile类提供两个构造函数来创建一块匿名共享内存，如下所示。

frameworks/base/core/java/android/os/MemoryFile.java

```
01 public class MemoryFile
02 {
03     ......
04
05     public MemoryFile(String name, int length) throws IOException {
06         mLength = length;
07         mFD = native_open(name, length);
08         mAddress = native_mmap(mFD, length, PROT_READ | PROT_WRITE);
09         mOwnsRegion = true;
10     }
11
12     ......
13
14     public MemoryFile(FileDescriptor fd, int length, String mode) throws IOException {
15         if (fd == null) {
16             throw new NullPointerException("File descriptor is null.");
17         }
18         if (!isMemoryFile(fd)) {
19             throw new IllegalArgumentException("Not a memory file.");
20         }
21         mLength = length;
22         mFD = fd;
23         mAddress = native_mmap(mFD, length, modeToProt(mode));
24         mOwnsRegion = false;
25     }
26
27     ......
28 }
```

第一个构造函数有两个参数name和length，分别用来描述要创建的匿名共享内存的名称和大小。第7行首先调用JNI方法native_open打开设备文件/dev/ashmem，即请求Ashmem驱动程序创建一块匿名共享内存，接着第8行调用JNI方法native_mmap将这块匿名共享内存映射到进程的地址空间，并且将得到的地址保存在成员变量mAddress中。

第二个构造函数有三个参数fd、length和mode，其中，参数fd是一个文件描述符，它指向一块匿名共享内存；参数length和mode用来描述这块匿名共享内存的大小和访问保护位。第18行调用成员函数isMemoryFile来验证文件描述符fd是否指向一块匿名共享内存。如果不是，那么参数fd就是一个非法参数，因此，第19行就抛出一个异常；否则，第23行就调用JNI方法native_mmap将参数fd所指向的一块匿名共享内存映射到进程的地址空间，并且将得到的地址保存在成员变量mAddress中。

一般来说，Server端应用程序首先使用第一个构造函数来创建一块匿名共享内存，接着再将这块匿名共享内存的文件描述符、大小以及访问保护位传递给Client端应用程序，以便它可以使用第二个构造函数将前面已经创建好的匿名共享内存映射到本进程的地址空间，这样Server端应用程序和Client端应用程序就可以共享同一块匿名共享内存了。

MemoryFile类的成员函数isMemoryFile的实现如下所示。

frameworks/base/core/java/android/os/MemoryFile.java

```
01 public class MemoryFile
02 {
03     ......
04
05     public static boolean isMemoryFile(FileDescriptor fd) throws IOException {
06         return (native_get_size(fd) >= 0);
07     }
08
09     ......
10 }
```

第6行调用了一个JNI方法native_get_size来获得文件描述符fd所指向的一块匿名共享内存的大小。如果获得的大小不是一个负数，那么就说明文件描述符fd指向的是一块匿名共享内存，这时候函数的返回值就等于true。

接下来，我们分别分析JNI方法native_get_size、native_open和native_mmap的实现。

JNI方法native_get_size的实现如下所示。

frameworks/base/core/jni/android_os_MemoryFile.cpp

```
01 static jint android_os_MemoryFile_get_size(JNIEnv* env, jobject clazz,
02         jobject fileDescriptor) {
03     int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
04     // Use ASHMEM_GET_SIZE to find out if the fd refers to an ashmem region.
05     // ASHMEM_GET_SIZE should succeed for all ashmem regions, and the kernel
06     // should return ENOTTY for all other valid file descriptors
07     int result = ashmem_get_size_region(fd);
08     if (result < 0) {
09         if (errno == ENOTTY) {
10             // ENOTTY means that the ioctl does not apply to this object,
11             // i.e., it is not an ashmem region.
12             return (jint) -1;
13         }
14         // Some other error, throw exception
15         jniThrowIOException(env, errno);
16         return (jint) -1;
17     }
18     return (jint) result;
19 }
```

第3行首先调用函数jniGetFDFromFileDescriptor将Java层的文件描述符fileDescriptor转换成一个C++层的文件描述符fd，接着第7行调用运行时库cutils提供的函数ashmem_get_size_region向Ashmem驱动程序发出一个IO控制命令ASHMEM_GET_SIZE，以便它可以返回与文件描述符fd对应的匿名共享内存的大小。如果文件描述符fd指向的不是一块匿名共享内存，那么在调用函数ashmem_get_size_region时，就不会进入到Ashmem驱动程序中，而是进入到其他的驱动程序中。由于其他的驱动程序没有实现IO控制命令ASHMEM_GET_SIZE，因此，它就会返回一个错误代码ENOTTY给调用者。这样，JNI方法native_get_size就可以判断出一个文件描述符是否指向了一块匿名共享内存。

JNI方法native_open的实现如下所示。

frameworks/base/core/jni/android_os_MemoryFile.cpp

```
01 static jobject android_os_MemoryFile_open(JNIEnv* env, jobject clazz, jstring name, jint length)
02 {
03     const char* namestr = (name ? env->GetStringUTFChars(name, NULL) : NULL);
04
05     int result = ashmem_create_region(namestr, length);
06
07     if (name)
08         env->ReleaseStringUTFChars(name, namestr);
09
10     if (result < 0) {
11         jniThrowException(env, "java/io/IOException", "ashmem_create_region failed");
12         return NULL;
13     }
14
15     return jniCreateFileDescriptor(env, result);
16 }
```

第5行调用运行时库cutils提供的函数ashmem_create_region来创建一块匿名共享内存。创建成功之后，就会得到一个C++层的文件描述符result，接着第15行调用函数jniCreateFileDescriptor将它转换成一个 Java层的文件描述，并且返回给调用者。

JNI方法native_mmap的实现如下所示。

frameworks/base/core/jni/android_os_MemoryFile.cpp

```
1 static jint android_os_MemoryFile_mmap(JNIEnv* env, jobject clazz, jobject fileDescriptor,
2         jint length, jint prot)
3 {
4     int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
5     jint result = (jint)mmap(NULL, length, prot, MAP_SHARED, fd, 0);
6     if (!result)
7         jniThrowException(env, "java/io/IOException", "mmap failed");
8     return result;
9 }
```

第4行首先调用函数jniGetFDFromFileDescriptor将Java层的文件描述符fileDescriptor转换成一个C++层的文件描述符fd，接着再调用函数mmap将它所指向的一块匿名共享内存映射到进程的地址空间，最后第8行将得到的地址result返回给调用者。

MemoryFile类的成员函数readBytes和writeBytes分别用来读取和写入一块匿名共享内存的内容，它们的实现如下所示。

frameworks/base/core/java/android/os/MemoryFile.java

```
01 public class MemoryFile
02 {
03     ......
04
05     public int readBytes(byte[] buffer, int srcOffset, int destOffset, int count)
06             throws IOException {
07         if (isDeactivated()) {
08             throw new IOException("Can't read from deactivated memory file.");
09         }
10         if (destOffset < 0 || destOffset > buffer.length || count < 0
11                 || count > buffer.length - destOffset
12                 || srcOffset < 0 || srcOffset > mLength
13                 || count > mLength - srcOffset) {
14             throw new IndexOutOfBoundsException();
15         }
16         return native_read(mFD, mAddress, buffer, srcOffset, destOffset, count, mAllowPurging);
17     }
18
19     public void writeBytes(byte[] buffer, int srcOffset, int destOffset, int count)
20             throws IOException {
21         if (isDeactivated()) {
22             throw new IOException("Can't write to deactivated memory file.");
23         }
24         if (srcOffset < 0 || srcOffset > buffer.length || count < 0
25                 || count > buffer.length - srcOffset
26                 || destOffset < 0 || destOffset > mLength
27                 || count > mLength - destOffset) {
28             throw new IndexOutOfBoundsException();
29         }
30         native_write(mFD, mAddress, buffer, srcOffset, destOffset, count, mAllowPurging);
31     }
32
33     ......
34 }
```

在读取或者写入一块匿名共享内存时，首先要保证这块匿名共享内存已经映射到进程的地址空间中，这是通过调用成员函数isDeactivated来判断的，它的实现如下所示。

frameworks/base/core/java/android/os/MemoryFile.java

```
01 public class MemoryFile
02 {
```

```
    ......

    private boolean isDeactivated() {
        return mAddress == 0;
    }

    ......
}
```

MemoryFile类的成员变量mAddress保存的是一块匿名共享内存的地址值。如果它的值等于0，那么就说明这块匿名共享内存尚未映射到进程的地址空间，因此，成员函数isDeactivated的返回值就等于true。

回到MemoryFile类的成员函数readBytes或者writeBytes中，前者最后调用JNI方法native_read将一块匿名共享内存的内容读入到缓冲区buffer中，而后者最后调用JNI方法将缓冲区buffer的内容写入到一块匿名共享内存中。

接下来，我们就一起分析JNI方法native_read和native_write的实现。这两个JNI方法是分别由C++层中的函数android_os_MemoryFile_read和android_os_MemoryFile_write来实现的，如下所示。

frameworks/base/core/jni/android_os_MemoryFile.cpp

```
static jint android_os_MemoryFile_read(JNIEnv* env, jobject clazz,
        jobject fileDescriptor, jint address, jbyteArray buffer, jint srcOffset, jint destOffset,
        jint count, jboolean unpinned)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    if (unpinned && ashmem_pin_region(fd, 0, 0) == ASHMEM_WAS_PURGED) {
        ashmem_unpin_region(fd, 0, 0);
        jniThrowException(env, "java/io/IOException", "ashmem region was purged");
        return -1;
    }

    env->SetByteArrayRegion(buffer, destOffset, count, (const jbyte *)address + srcOffset);

    if (unpinned) {
        ashmem_unpin_region(fd, 0, 0);
    }
    return count;
}

static jint android_os_MemoryFile_write(JNIEnv* env, jobject clazz,
        jobject fileDescriptor, jint address, jbyteArray buffer, jint srcOffset, jint destOffset,
        jint count, jboolean unpinned)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    if (unpinned && ashmem_pin_region(fd, 0, 0) == ASHMEM_WAS_PURGED) {
        ashmem_unpin_region(fd, 0, 0);
        jniThrowException(env, "java/io/IOException", "ashmem region was purged");
        return -1;
    }

    env->GetByteArrayRegion(buffer, srcOffset, count, (jbyte *)address + destOffset);

    if (unpinned) {
        ashmem_unpin_region(fd, 0, 0);
    }
    return count;
}
```

MemoryFile类有一个成员变量mAllowPurging，它是一个布尔变量。如果它的值等于true，那么就表示每次访问MemoryFile类内部的匿名共享内存之后，都要将它解锁，以便它可以被内存管理系统回收。

在函数android_os_MemoryFile_read和android_os_MemoryFile_write中，参数unpinned 的值就对应于MemoryFile类的成员变量mAllowPurging的值；因此，当它等于true时，这两个JNI方法就需要调用运行时库cutils提供的函数ashmem_pin_region来判断MemoryFile类内部的匿名共享内存是否已经被内存管理系统回收。如果函数ashmem_pin_region的返回值等于ASHMEM_WAS_PURGED，就说明这块匿名共享内存已经被内存管理系统回收了，这时候这两个JNI方法就会返回一个错误代码-1给调用者，表示它们不能够再访问MemoryFile类内部的匿名共享内存了。

如果MemoryFile类内部的匿名共享内存还没有被内存管理系统回收，那么第12行和第31行就可以正常访问它的内容了。访问结束之后，如果发现参数unpinned的值等于true，那么第15行和第34行还需要将这块匿名共享内存解锁，以便在系统内存不足时，内存管理系统可以将它回收。

至此，匿名共享内存的Java访问接口MemoryFile就分析完成了。接下来，我们通过一个应用实例来说明它的使用方法。

6.4.2　应用实例

在本节中，我们将创建一个Android应用程序Ashmem，它由一个Service组件Server和一个Activity组件Client组成。Server组件运行在一个独立的进程中，它内部有一个内存访问服务MemoryService，后者通过MemoryFile类创建了一块匿名共享内存。Client组件运行在另外一个进程中，它会将内存访问服务MemoryService创建的那块匿名共享内存映射到本进程的地址空间，以便可以访问它的内容，从而可以和Server组件共享同一块匿名共享内存。

应用程序Ashmem的目录结构如下：

```
~/Android/packages/experimental/Ashmem
----AndroidManifest.xml
----Android.mk
----src
    ----shy/luo/ashmem
        ----IMemoryService.java
        ----MemoryService.java
        ----Server.java
        ----Client.java
----res
    ----layout
        ----main.xml
    ----values
        ----strings.xml
    ----drawable
        ----icon.png
```

应用程序Ashmem包含了一个源代码目录src、一个资源目录res、一个程序配置文件AndroidManifest.xml和一个编译脚本文件Android.mk。接下来，我们就分别介绍每一个文件的内容。

IMemoryService.java

```
001 package shy.luo.ashmem;
002
003 import android.util.Log;
004 import android.os.IInterface;
005 import android.os.Binder;
006 import android.os.IBinder;
007 import android.os.Parcel;
008 import android.os.ParcelFileDescriptor;
009 import android.os.RemoteException;
010
```

```
011 public interface IMemoryService extends IInterface {
012         public static abstract class Stub extends Binder implements IMemoryService {
013                 private static final String DESCRIPTOR = "shy.luo.ashmem.IMemoryService";
014
015                 public Stub() {
016                         attachInterface(this, DESCRIPTOR);
017                 }
018
019                 public static IMemoryService asInterface(IBinder obj) {
020                         if (obj == null) {
021                                 return null;
022                         }
023
024                         IInterface iin = (IInterface)obj.queryLocalInterface(DESCRIPTOR);
025                         if (iin != null && iin instanceof IMemoryService) {
026                                 return (IMemoryService)iin;
027                         }
028
029                         return new IMemoryService.Stub.Proxy(obj);
030                 }
031
032                 public IBinder asBinder() {
033                         return this;
034                 }
035
036                 @Override
037                   public boolean onTransact(int code, Parcel data, Parcel reply, int flags)
throws android.os.RemoteException {
038                         switch (code) {
039                         case INTERFACE_TRANSACTION: {
040                                 reply.writeString(DESCRIPTOR);
041                                 return true;
042                         }
043                         case TRANSACTION_getFileDescriptor: {
044                                 data.enforceInterface(DESCRIPTOR);
045
046                                 ParcelFileDescriptor result = this.getFileDescriptor();
047                                 reply.writeNoException();
048                                 if (result != null) {
049                                         reply.writeInt(1);
050                                         result.writeToParcel(reply, 0);
051                                 } else {
052                                         reply.writeInt(0);
053                                 }
054
055                                 return true;
056                         }
057
058                         case TRANSACTION_setValue: {
059                                 data.enforceInterface(DESCRIPTOR);
060
061                                 int val = data.readInt();
062                                 setValue(val);
063                                 reply.writeNoException();
064
065                                 return true;
066                         }
067                         }
068
069                         return super.onTransact(code, data, reply, flags);
070                 }
071
072                 private static class Proxy implements IMemoryService {
```

```
073                         private IBinder mRemote;
074
075                         Proxy(IBinder remote) {
076                                 mRemote = remote;
077                         }
078
079                         public IBinder asBinder() {
080                                 return mRemote;
081                         }
082
083                         public String getInterfaceDescriptor() {
084                                 return DESCRIPTOR;
085                         }
086
087                         public ParcelFileDescriptor getFileDescriptor() throws RemoteException {
088                                 Parcel data = Parcel.obtain();
089                                 Parcel reply = Parcel.obtain();
090
091                                 ParcelFileDescriptor result;
092                                 try {
093                                         data.writeInterfaceToken(DESCRIPTOR);
094
095                                         mRemote.transact(Stub.TRANSACTION_getFileDescriptor,
data, reply, 0);
096                                         reply.readException();
097                                         if (0 != reply.readInt()) {
098                                              result = ParcelFileDescriptor.CREATOR.createFromParcel
(reply);
099                                         } else {
100                                              result = null;
101                                         }
102                                 } finally {
103                                         reply.recycle();
104                                         data.recycle();
105                                 }
106
107                                 return result;
108                         }
109
110                         public void setValue(int val) throws RemoteException {
111                                 Parcel data = Parcel.obtain();
112                                 Parcel reply = Parcel.obtain();
113
114                                 try {
115                                         data.writeInterfaceToken(DESCRIPTOR);
116                                         data.writeInt(val);
117
118                                         mRemote.transact(Stub.TRANSACTION_setValue, data, reply, 0);
119                                         reply.readException();
120                                 } finally {
121                                         reply.recycle();
122                                         data.recycle();
123                                 }
124                         }
125                 }
126
127                 static final int TRANSACTION_getFileDescriptor = IBinder.FIRST_CALL_TRANSACTION + 0;
128                 static final int TRANSACTION_setValue = IBinder.FIRST_CALL_TRANSACTION + 1;
129
130         }
131
132         public ParcelFileDescriptor getFileDescriptor() throws RemoteException;
133         public void setValue(int val) throws RemoteException;
134 }
```

这个文件定义了MemoryService的服务接口IMemoryService、Java本地服务类IMemoryService.Stub和Java服务代理对象类IMemoryService.Stub.Proxy。

IMemoryService接口定义了两个成员函数getFileDescriptor和setValue，其中，getFileDescriptor用来获得一块匿名共享内存的文件描述符，而setValue用来往这块匿名共享内存中写入一个整数。IMemoryFile接口的成员函数getFileDescriptor的返回值是一个ParcelFileDescriptor对象，它也是用来描述一个文件描述符的，不过，它描述的文件描述符是用来从一个进程传递到另一个进程的。

IMemoryService.Stub是一个Java本地服务类，内存访问服务MemoryService必须以它为父类。成员函数onTransact用来处理类型为TRANSACTION_getFileDescriptor或者TRANSACTION_setValue的进程间通信请求；其中，第46行调用由其子类重写的成员函数getFileDescriptor来处理类型为TRANSACTION_getFileDescriptor的进程间通信请求，而第62行调用由其子类重写的成员函数setValue来处理类型为TRANSACTION_setValue的进程间通信请求。静态成员函数asInterface用来将一个Java服务代理对象封装成一个实现了IMemoryService接口的Java服务代理对象，即一个IMemoryService.Stub.Proxy对象。

IMemoryService.Stub.Proxy是一个Java服务代理对象类，用来描述一个内存访问服务MemoryService的Java代理对象。成员变量mRemote指向一个Java服务代理对象，用来向运行在另外一个进程中的MemoryService服务发送进程间通信请求。成员函数getFileDescriptor用来向运行在另外一个进程中的MemoryService服务发送一个类型为TRANSACTION_getFileDescriptor的进程间通信请求，以便可以获得该MemoryService服务内部的一块匿名共享内存的文件描述符。成员函数setValue用来向运行在另外一个进程中的MemoryService服务发送一个类型为TRANSACTION_setValue的进程间通信请求，以便可以向该MemoryService服务内部的一块匿名共享内存中写入一个整数值。

MemoryService.java

```
01 package shy.luo.ashmem;
02
03 import java.io.FileDescriptor;
04 import java.io.IOException;
05
06 import android.os.Parcel;
07 import android.os.MemoryFile;
08 import android.os.ParcelFileDescriptor;
09 import android.util.Log;
10
11 public class MemoryService extends IMemoryService.Stub {
12         private final static String LOG_TAG = "shy.luo.ashmem.MemoryService";
13         private MemoryFile file = null;
14
15         public MemoryService() {
16                 try {
17                         file = new MemoryFile("Ashmem", 4);
18                         setValue(0);
19                 }
20                 catch(IOException ex) {
21                         Log.i(LOG_TAG, "Failed to create memory file.");
22                         ex.printStackTrace();
23                 }
24         }
25
26         public ParcelFileDescriptor getFileDescriptor() {
27                 ParcelFileDescriptor pfd = null;
28
29                 try {
30                         pfd = file.getParcelFileDescriptor();
31                 } catch(IOException ex) {
```

```
32                          Log.i(LOG_TAG, "Failed to get file descriptor.");
33                          ex.printStackTrace();
34                  }
35
36                  return pfd;
37          }
38
39          public void setValue(int val) {
40                  if(file == null) {
41                          return;
42                  }
43
44                  byte[] buffer = new byte[4];
45                  buffer[0] = (byte)((val >>> 24) & 0xFF);
46                  buffer[1] = (byte)((val >>> 16) & 0xFF);
47                  buffer[2] = (byte)((val >>> 8) & 0xFF);
48                  buffer[3] = (byte)(val & 0xFF);
49
50                  try {
51                          file.writeBytes(buffer, 0, 0, 4);
52                          Log.i(LOG_TAG, "Set value " + val + " to memory file. ");
53                  }
54                  catch(IOException ex) {
55                          Log.i(LOG_TAG, "Failed to write bytes to memory file.");
56                          ex.printStackTrace();
57                  }
58          }
59  }
```

这个文件定义了一个Java服务MemoryService，它继承了IMemoryService.Stub类，并且实现了IMemoryService接口的成员函数getFileDescriptor和setValue。

在MemoryService类的构造函数中，第17行创建了一个MemoryFile对象，并且将它保存在成员变量file中。从前面6.4.1节的内容可以知道，MemoryFile对象file在创建的过程中，会请求Ashmem驱动程序创建一块匿名共享内存，这块匿名共享内存的名称和大小分别为“Ashmem”和4。

在MemoryService类的成员函数getFileDescriptor中，第30行调用成员变量file的成员函数getParcelFileDescriptor来获得其内部的一块匿名共享内存的文件描述符，最后第36行将它返回给调用者。

在MemoryService类的成员函数setValue中，第44行到第48行代码首先将整型参数val写入到一个字节数组buffer中，接着调用成员变量file的成员函数writeBytes将它的内容写入到其内部的一块匿名共享内存中。

Server.java

```
01 package shy.luo.ashmem;
02
03 import android.app.Service;
04 import android.content.Intent;
05 import android.os.IBinder;
06 import android.util.Log;
07 import android.os.ServiceManager;
08
09 public class Server extends Service {
10     private final static String LOG_TAG = "shy.luo.ashmem.Server";
11     private MemoryService memoryService = null;
12
13     @Override
14     public IBinder onBind(Intent intent) {
15         return null;
16     }
17
```

```
18      @Override
19      public void onCreate() {
20          memoryService = new MemoryService();
21
22          try {
23              ServiceManager.addService("AnonymousSharedMemory", memoryService);
24              Log.i(LOG_TAG, "Succeed to add memory service.");
25          } catch (RuntimeException ex) {
26              Log.i(LOG_TAG, "Failed to add Memory Service.");
27              ex.printStackTrace();
28          }
29      }
30
31      @Override
32      public void onStart(Intent intent, int startId) {
33          Log.i(LOG_TAG, "Start Memory Service.");
34      }
35
36      @Override
37      public void onDestroy() {
38          Log.i(LOG_TAG, "Destroy Memory Service.");
39      }
40  }
```

这个文件定义了一个Service组件Server。

注意

这是一个标准的Android应用程序组件，与Binder进程间通信机制中的Service组件是两个不同的概念。

Server组件运行在一个独立的进程中，它在启动时，第20行首先会创建一个内存访问服务MemoryService，接着第23行将这个服务注册到Service Manager中。

内存访问服务MemoryService注册到Service Manager中的名称被指定为“AnonymousSharedMemory”，这样，其他进程就可以通过这个名称来获得MemoryService服务的一个Java代理对象。

Client.java

```
001 package shy.luo.ashmem;
002
003 import java.io.FileDescriptor;
004 import java.io.IOException;
005
006 import android.app.Activity;
007 import android.content.Intent;
008 import android.os.Bundle;
009 import android.os.MemoryFile;
010 import android.os.ParcelFileDescriptor;
011 import android.os.ServiceManager;
012 import android.os.RemoteException;
013 import android.util.Log;
014 import android.view.View;
015 import android.view.View.OnClickListener;
016 import android.widget.Button;
017 import android.widget.EditText;
018
019 public class Client extends Activity implements OnClickListener {
020         private final static String LOG_TAG = "shy.luo.ashmem.Client";
021
022         IMemoryService memoryService = null;
023         MemoryFile memoryFile = null;
024
025         private EditText valueText = null;
```

```
026         private Button readButton = null;
027         private Button writeButton = null;
028         private Button clearButton = null;
029 
030         @Override
031         public void onCreate(Bundle savedInstanceState) {
032                 super.onCreate(savedInstanceState);
033                 setContentView(R.layout.main);
034 
035                 IMemoryService ms = getMemoryService();
036                 if(ms == null) {
037                         Log.i(LOG_TAG, "start shy.luo.ashmem.server");
038                         startService(new Intent("shy.luo.ashmem.server"));
039                 } else {
040                         Log.i(LOG_TAG, "Memory Service has started.");
041                 }
042 
043                 valueText = (EditText)findViewById(R.id.edit_value);
044                 readButton = (Button)findViewById(R.id.button_read);
045                 writeButton = (Button)findViewById(R.id.button_write);
046                 clearButton = (Button)findViewById(R.id.button_clear);
047 
048                 readButton.setOnClickListener(this);
049                 writeButton.setOnClickListener(this);
050                 clearButton.setOnClickListener(this);
051 
052                 Log.i(LOG_TAG, "Client Activity Created.");
053         }
054 
055         @Override
056         public void onClick(View v) {
057                 if(v.equals(readButton)) {
058                         int val = 0;
059 
060                         MemoryFile mf = getMemoryFile();
061                         if(mf != null) {
062                                 try {
063                                         byte[] buffer = new byte[4];
064                                         mf.readBytes(buffer, 0, 0, 4);
065 
066                                         val = (buffer[0] << 24) | ((buffer[1] & 0xFF) << 16)
| ((buffer[2] & 0xFF) << 8) | (buffer[3] & 0xFF);
067                                 } catch(IOException ex) {
068                                         Log.i(LOG_TAG, "Failed to read bytes from memory file.");
069                                         ex.printStackTrace();
070                                 }
071                         }
072 
073                         String text = String.valueOf(val);
074                         valueText.setText(text);
075                 } else if(v.equals(writeButton)) {
076                         String text = valueText.getText().toString();
077                         int val = Integer.parseInt(text);
078 
079                         IMemoryService ms = getMemoryService();
080                         if(ms != null) {
081                                 try {
082                                         ms.setValue(val);
083                                 } catch(RemoteException ex) {
084                                         Log.i(LOG_TAG, "Failed to set value to memory service.");
085                                         ex.printStackTrace();
086                                 }
087                         }
```

```
088                 } else if(v.equals(clearButton)) {
089                         String text = "";
090                         valueText.setText(text);
091                 }
092         }
093
094         private IMemoryService getMemoryService() {
095                 if(memoryService != null) {
096                         return memoryService;
097                 }
098
099                 memoryService = IMemoryService.Stub.asInterface(
100                                         ServiceManager.getService("AnonymousSharedMemory"));
101
102                  Log.i(LOG_TAG, memoryService != null ? "Succeed to get memeory service." :
"Failed to get memory service.");
103
104                 return memoryService;
105         }
106
107         private MemoryFile getMemoryFile() {
108                 if(memoryFile != null) {
109                         return memoryFile;
110                 }
111
112                 IMemoryService ms = getMemoryService();
113                 if(ms != null) {
114                         try {
115                                 ParcelFileDescriptor pfd = ms.getFileDescriptor();
116                                 if(pfd == null) {
117                                         Log.i(LOG_TAG, "Failed to get memory file descriptor.");
118                                         return null;
119                                 }
120
121                                 try {
122                                         FileDescriptor fd = pfd.getFileDescriptor();
123                                         if(fd == null) {
124                                                 Log.i(LOG_TAG, "Failed to get memeory file descriptor.");
125                                                 return null;
126                                         }
127
128                                         memoryFile = new MemoryFile(fd, 4, "r");
129                                 } catch(IOException ex) {
130                                         Log.i(LOG_TAG, "Failed to create memory file.");
131                                         ex.printStackTrace();
132                                 }
133                         } catch(RemoteException ex) {
134                                 Log.i(LOG_TAG, "Failed to get file descriptor from memory service.");
135                                 ex.printStackTrace();
136                         }
137                 }
138
139                 return memoryFile;
140         }
141 }
```

这个文件定义了一个Activity组件Client，用来描述应用程序Ashmem的主界面。

Client组件在启动时，第38行会调用成员函数startService来启动应用程序Ashmem的Server组件。前面提到，这个Server组件是运行在一个独立的进程中的，它在启动时，会创建一块匿名共享内存，并且将用来管理这块匿名共享内存的MemoryService服务注册到Service Manager中。

注意

应用程序Ashmem的Server组件是被隐式启动的，Client组件在调用成员函数startService时，只要指定Server组件的Action名称“start shy.luo.ashmem.server”就可以将它隐式启动起来。

Client组件的界面上有三个按钮Read、Write和Clear，以及一个文本框EditText。当点击Read按钮时，Client组件便会将在MemoryService服务中创建的匿名共享内存的内容读取出来，并且显示在文本框EditText中。当点击Write按钮时，Client组件便会将输入在文本框EditText中的整数写入到在MemoryService服务中创建的匿名共享内存中。当点击Clear按钮时，Client组件便会清空文本框EditText的内容。

Client组件的成员函数getMemoryService用来获取一个MemoryService服务代理对象。第一次调用Client组件的成员函数getMemoryService时，Client组件会通过Service Manager来获得MemoryService服务的一个Java代理对象，并且将它保存在成员变量memoryService中。

Client组件的成员函数getMemoryFile用来获得一个MemoryFile对象。第一次调用Client组件的成员函数getMemoryFile时，Client组件会通过内部的成员变量memoryService来请求MemoryService服务返回其内部的一块匿名共享内存的文件描述符fd。有了文件描述符fd之后，Client组件就可以在内部创建一个MemoryFile对象，并且将这个MemoryFile对象保存在成员变量memoryFile中。从前面6.4.1节的内容可以知道，MemoryFile对象memoryFile在创建的过程中，会将与文件描述符fd对应的匿名共享内存映射到本进程的地址空间，即将在MemoryService服务中创建的那块匿名共享内存映射到本进程的地址空间，这样就可以在本进程中直接访问这块匿名共享内存的内容了。

注意

当点击界面上的Read按钮时，Client组件是通过调用其内部的MemoryFile对象memoryFile的成员函数readBytes来访问在MemoryService服务中创建的匿名共享内存的内容的，即通过本进程的地址空间来访问这块匿名共享内存的内容；而当点击界面上的Write按钮时，Client组件是通过调用其内部的MemoryService服务代理对象memoryService的成员函数setValue来将一个整数值写入到在MemoryService服务中创建的匿名共享内存中的。

main.xml

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
03     android:orientation="vertical"
04     android:layout_width="fill_parent"
05     android:layout_height="fill_parent">
06     <LinearLayout
07         android:layout_width="fill_parent"
08         android:layout_height="wrap_content"
09         android:orientation="vertical"
10         android:gravity="center">
11         <TextView
12                 android:layout_width="wrap_content"
13                 android:layout_height="wrap_content"
14                 android:text="@string/value">
15         </TextView>
16         <EditText
17                 android:layout_width="fill_parent"
18                 android:layout_height="wrap_content"
19                 android:id="@+id/edit_value"
20                 android:hint="@string/hint">
21         </EditText>
```

```
22      </LinearLayout>
23      <LinearLayout
24          android:layout_width="fill_parent"
25          android:layout_height="wrap_content"
26          android:orientation="horizontal"
27          android:gravity="center">
28          <Button
29                  android:id="@+id/button_read"
30                  android:layout_width="wrap_content"
31                  android:layout_height="wrap_content"
32                  android:text="@string/read">
33          </Button>
34          <Button
35                  android:id="@+id/button_write"
36                  android:layout_width="wrap_content"
37                  android:layout_height="wrap_content"
38                  android:text="@string/write">
39          </Button>
40          <Button
41                  android:id="@+id/button_clear"
42                  android:layout_width="wrap_content"
43                  android:layout_height="wrap_content"
44                  android:text="@string/clear">
45          </Button>
46      </LinearLayout>
47 </LinearLayout>
```

这是Client组件的界面配置文件，在屏幕中间显示一个EditText控件和三个Button控件。

strings.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3      <string name="app_name">Ashmem</string>
4      <string name="value">Value</string>
5      <string name="hint">Please input a value...</string>
6      <string name="read">Read</string>
7      <string name="write">Write</string>
8      <string name="clear">Clear</string>
9 </resources>
```

这是应用程序Ashmem的字符串资源文件，定义了在应用程序中用到的各个字符串。

icon.png

这是应用程序Ashmem的图标，可以根据需要来放置不同的图片文件。

AndroidManifest.xml

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
03       package="shy.luo.ashmem"
04       android:sharedUserId="android.uid.system"
05       android:versionCode="1"
06       android:versionName="1.0">
07       <application android:icon="@drawable/icon" android:label="@string/app_name">
08             <activity android:name=".Client"
09                   android:label="@string/app_name">
10                   <intent-filter>
11                         <action android:name="android.intent.action.MAIN" />
12                         <category android:name="android.intent.category.LAUNCHER" />
13                   </intent-filter>
14              </activity>
```

```
15              <service
16                   android:enabled="true"
17                   android:name=".Server"
18                   android:process="shy.luo.ashmem.server" >
19                   <intent-filter>
20                         <action android:name="shy.luo.ashmem.server"/>
21                         <category android:name="android.intent.category.DEFAULT"/>
22                    </intent-filter>
23              </service>
24       </application>
25 </manifest>
```

这是应用程序Ashmem的配置文件。由于在程序中使用到了Activity组件Client和Service组件Server，因此，我们需要在这里对它们进行配置。

注意

我们将Client组件作为应用程序Ashmem的根Activity；同时，在配置Server组件时，我们将它的android:process属性设置为“shy.luo.ashmem.server”，即告诉Android系统，将这个Server组件运行在一个名称为“shy.luo.ashmem.server”的独立进程中。

由于我们需要在应用程序Ashmem中注册MemoryService服务，因此，需要为应用程序Ashmem申请System用户权限，即将应用程序Ashmem的android:sharedUserId属性设置为“android.uid.system”，以便应用程序Ashmem可以获得注册服务的权限。

Android.mk

```
1 LOCAL_PATH:= $(call my-dir)
2 include $(CLEAR_VARS)
3 LOCAL_MODULE_TAGS := optional
4 LOCAL_SRC_FILES += $(call all-subdir-java-files)
5 LOCAL_PACKAGE_NAME := Ashmem
6 LOCAL_CERTIFICATE := platform
7 include $(BUILD_PACKAGE)
```

这是应用程序Ashmem的编译脚本文件。由于我们需要在应用程序Ashmem中注册服务，因此，就需要使用平台类型的证书来编译应用程序Ashmem，即将变量LOCAL_CERTIFICATE的值设置为“platform”，以便应用程序Ashmem可以获得注册服务的权限。

上述文件都准备好之后，我们就可以对应用程序Ashmem进行编译和打包了。

```
USER@MACHINE:~/Android$ mmm ./packages/experimental/Ashmem/
USER@MACHINE:~/Android$ make snod
```

编译成功之后，就可以在out/target/product/gerneric/system/app目录下看到应用程序文件Ashmem.apk了；打包成功之后，应用程序Ashmem就包含在out/target/product/gerneric目录下的Android系统镜像文件system.img中了。

最后，我们使用新得到的系统镜像文件system.img来启动Android模拟器。

```
USER@MACHINE:~/Android$ emulator
```

Android模拟器运行起来之后，我们就可以在应用程序启动器中启动应用程序Ashmem了，它的界面如图6–8所示。

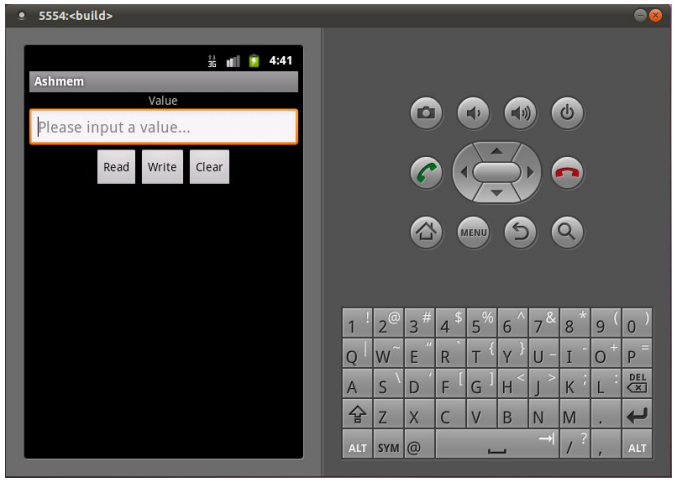


图6-8　应用程序Ashmem的界面

在这个界面中，我们可以通过点击Read和Write按钮来验证应用程序Ashmem的两个进程是否成功地共享了同一块匿名共享内存。

至此，匿名共享内存的C++和Java访问接口就分析完成了。接下来，我们继续分析匿名共享内存在两个不同的进程当中进行共享的原理。

6.5　匿名共享内存的共享原理

在Android系统中，每一块匿名共享内存都是使用一个文件描述符来描述的，而这个文件描述符是通过打开设备文件/dev/ashmem获得的。当两个进程需要共享一块匿名共享内存时，只要把它的文件描述符从一个进程传递给别外一个进程即可。

我们知道，在Linux系统中，文件描述符其实就是一个整数，它只在进程范围内有效，即值相等的两个文件描述符在两个不同的进程中具有不同的含义。在Linux内核中，每一个文件描述符都对应有一个文件结构体（struct file）。文件结构体是一个内核对象，每一个打开的文件都有一个对应的文件结构体。文件描述符、文件结构体和文件的关系如图6-9所示。

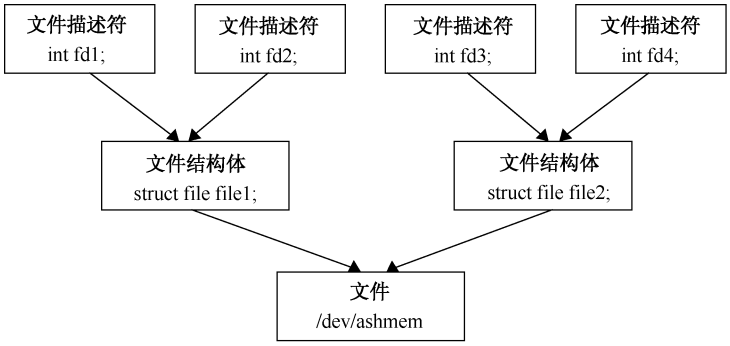


图6-9　文件描述符、文件结构体和文件的关系

不同的文件描述符可以对应于同一个文件结构体，而不同的文件结构体也可以对应于同一个文件。当应用程序调用函数open来打开一个文件时，文件系统就会为该文件创建一个文件结构体和一个文件描述符，最后将这个文件描述符返回给应用程序。

由于应用程序打开设备文件/dev/ashmem时，Ashmem驱动程序会为它在内核中创建一块匿名共享内存。因此，文件描述符、文件结构体和匿名共享内存的关系就如图6-10所示。

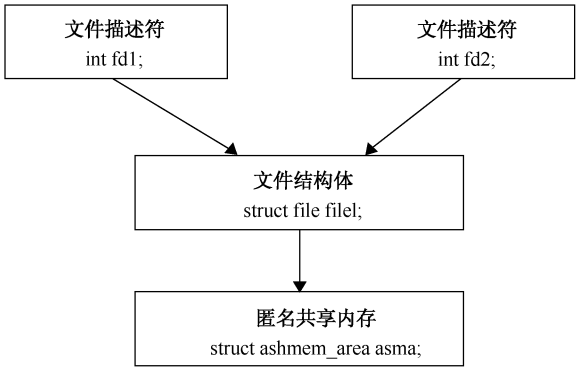


图6-10　文件描述符、文件结构体和匿名共享内存的关系

匿名共享内存能够在两个不同的进程中共享的奥妙就在于，这两个进程分别有一个文件描述符fd1和fd2，它们指向了同一个文件结构体file1，而这个文件结构体又指向了一块匿名共享内存asma。这时候，如果这两个进程的文件描述符fd1和fd2分别被映射到各自的地址空间，那么它们就会把同一块匿名共享内存映射到各自的地址空间，从而实现在两个不同的进程中共享同一块匿名共享内存。

现在的问题就是如何让两个位于不同进程中的文件描述符fd1和fd2指向同一个用来描述匿名共享内存asma的文件结构体file1。

假设进程p1首先调用函数open来打开设备文件/dev/ashmem，这样它就得到了一块匿名共享内存asma、一个文件结构体file1和一个文件描述符fd1。接着进程p2通过Binder进程间通信机制请求进程p1将文件描述符fd1返回给它，进程p1要通过Binder驱动程序将文件描述符fd1返回给进程p2。由于文件描述符fd1只在进程p1中有效，因此，Binder驱动程序就不能直接将文件描述符fd1返回给进程p2。这时候Binder驱动程序就会在进程p2中创建一个新的文件描述符fd2，使得它也指向文件结构体file1，最后再将文件描述符fd2返回给进程p2。这样，文件描述符fd1和fd2就指向同一个文件结构体file1了，即指向了同一块匿名共享内存。

在前面的6.4.2节中，Client组件通过其内部的一个MemoryService代理对象的成员函数getFileDescriptor来请求运行在另外一个进程中的MemoryService服务返回其内部的一块匿名共享内存的文件描述符，这个过程如图6-11所示。

第1步到第4步是Client组件请求MemoryService服务返回其内部的匿名共享内存的文件描述符的过程，而第5步到第8步是MemoryService服务返回其内部的匿名共享内存的文件描述符给Client组件的过程。

在第5步中，MemoryService服务将内部的匿名共享内存的文件描述符封装成一个ParcelFileDescriptor对象，然后把它从Java层传输到C++层。到了第6步时，这个正在传输的ParcelFileDescriptor对象首先被转换为一个类型为BINDER_TYPE_FD的flat_binder_object结构体，然后传输给Binder驱动程序。到了第7步时，Binder驱动程序就会对第6步传输过来的flat_binder_object结构体进行处理，然后再返回给Client组件。

接下来，我们就分析Binder驱动程序是如何处理进程间通信数据中类型为BINDER_TYPE_FD的flat_binder_object结构体的。

Binder驱动程序是在函数binder_transaction中处理进程间通信数据中类型为BINDER_TYPE_FD的flat_binder_object结构体的，如下所示。

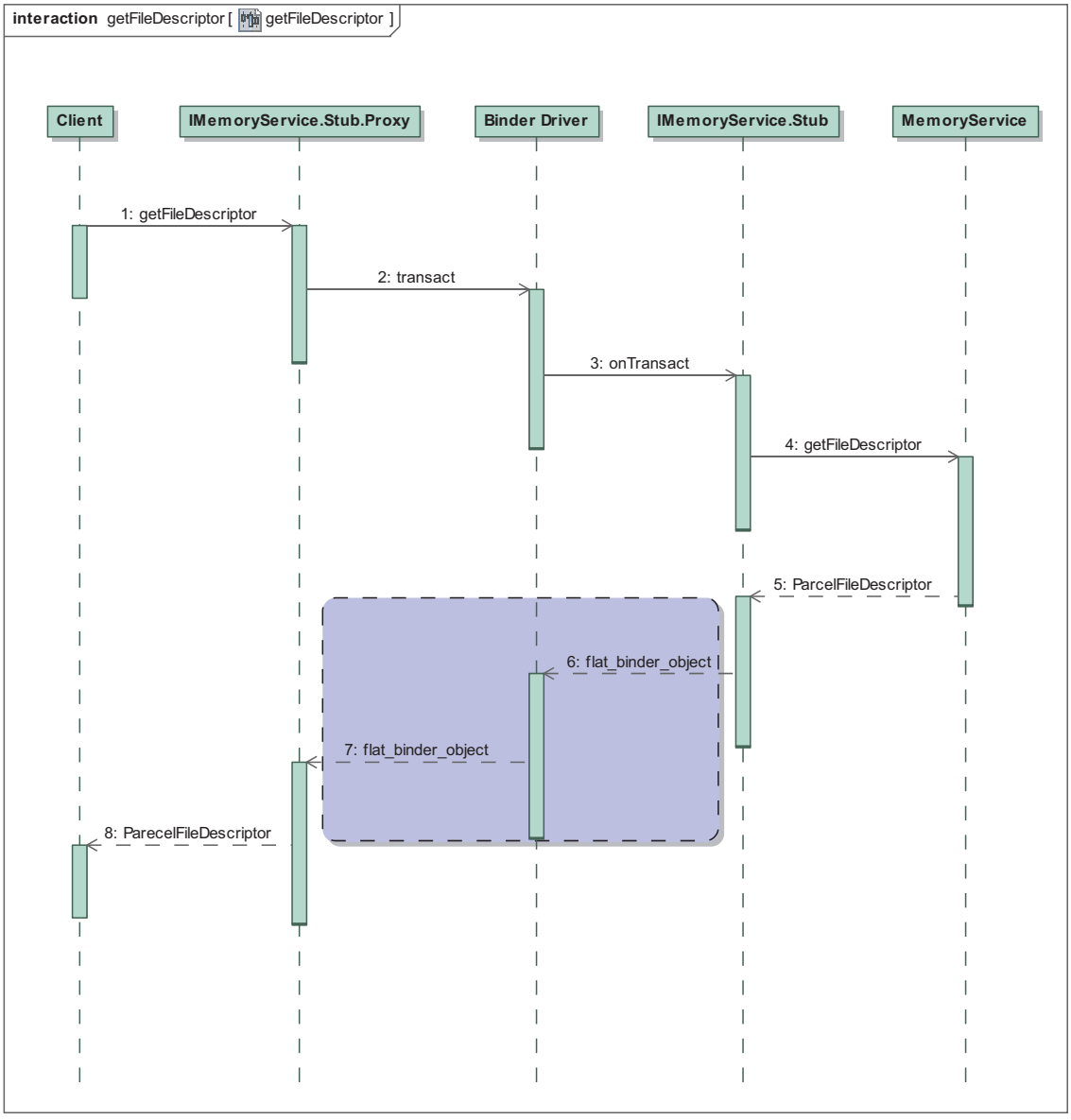


图6-11　匿名共享内存的文件描述符的传递过程

kernel/goldfish/drivers/staging/android/binder.c

```
01 static void
02 binder_transaction(struct binder_proc *proc, struct binder_thread *thread,
03     struct binder_transaction_data *tr, int reply)
04 {
05     ......
06
07     for (; offp < off_end; offp++) {
08         struct flat_binder_object *fp;
09         ......
10         fp = (struct flat_binder_object *)(t->buffer->data + *offp);
11         switch (fp->type) {
12         ......
13         case BINDER_TYPE_FD: {
14             int target_fd;
15             struct file *file;
16
17             ......
```

```
18
19              file = fget(fp->handle);
20              if (file == NULL) {
21                  ......
22                  return_error = BR_FAILED_REPLY;
23                  goto err_fget_failed;
24              }
25              target_fd = task_get_unused_fd_flags(target_proc, O_CLOEXEC);
26              if (target_fd < 0) {
27                  fput(file);
28                  return_error = BR_FAILED_REPLY;
29                  goto err_get_unused_fd_failed;
30              }
31              task_fd_install(target_proc, target_fd, file);
32              ......
33              fp->handle = target_fd;
34          } break;
35
36          ......
37          }
38      }
39
40      ......
41  }
```

第10行首先从进程间通信数据中获得一个类型为BINDER_TYPE_FD的flat_binder_object结构体fp，它的成员变量handle就是用来描述一个文件描述符的。

注意

这个文件描述符只在源进程中有效。接着第19行调用函数fget来获得与文件描述符fp->handle对应的文件结构体file。

第25行调用函数task_get_unused_fd_flags在目标进程target_proc中获得一个未使用的文件描述符target_fd，接着第31行调用函数task_fd_install将它和文件结构体file关联起来。这样，源进程中的文件描述符fp->handle和目标进程中的文件描述符target_fd就指向了同一个文件结构体，即指向了同一块匿名共享内存。

最后，第33行将flat_binder_object结构体fp的成员变量handle的值设置为文件描述符target_fd，以便可以将它返回给目标进程，即返回给Client组件。

Client组件从Binder驱动程序中获得了flat_binder_object结构体fp之后，首先将它封装成一个ParcelFileDescriptor对象，然后再将它转换成一个FileDescriptor对象，最后就可以使用这个FileDescriptor对象来创建一个MemoryFile对象，即将MemoryService服务内部的匿名共享内存映射到Client组件所在的进程的地址空间，从而达到了在不同的进程中共享同一块匿名共享内存的目的。

第3篇　Android应用程序框架

移动平台的核心竞争力是第三方应用程序的数量和质量。第三方应用程序的数量和质量除依赖于第三方的产品能力外，也依赖于移动平台本身所提供的应用程序框架。优秀的应用程序框架能够帮助第三方开发出大量高质量的应用程序，因此，在本篇中我们将重点分析Android平台的应用程序框架。

第7章到第10章分析Android应用程序的四大组件Activity、Service、Broadcast Receiver和Content Provider的实现原理；第7章和第8章从启动的角度来分析Activity组件和Service组件的实现原理；第9章从广播发送和接收的角度来分析Broadcast Receiver组件的实现原理；第10章从数据共享的角度来分析Content Provider组件的实现原理。

第11章分析Zygote和System进程的启动过程；第12章分析Android应用程序进程的启动过程。Zygote进程负责创建System进程和应用程序进程，System进程负责启动系统服务，而应用程序进程负责运行应用程序组件Activity、Service、Broadcast Receiver和Content Provider，并且为它们提供进程间通信机制。

第13章分析Android应用程序的消息处理机制，包括消息的循环、发送和处理过程；第14章分析Android应用程序键盘消息机制，包括键盘消息的发送和处理过程；第15章分析Android应用程序的线程与消息的关系。

第16章分析Android应用程序的安装和显示过程。系统在启动时，会扫描系统中的特定目录，然后自动安装里面的Android应用程序。系统在启动完成后，会启动一个Home应用程序来显示安装在系统中的Android应用程序。

第 7 章

Activity组件的启动过程

Activity是Android应用程序的四大组件之一，它负责管理Android应用程序的用户界面。一个应用程序一般会包含若干个Activity组件，每一个Activity组件负责一个用户界面的展现，它们可能运行在同一个进程中，也可能运行在不同的进程中。运行在不同进程中的Activity组件通过Binder进程间通信机制来协作完成应用程序的功能。由此可见，Android应用程序的组件化程度是非常高的，它可以将运行在不同进程中的Activity组件组织在一起形成自己的功能。在本章中，我们通过分析Activity组件的启动过程来理解它们在Android应用程序中的组织形式。

从应用程序的角度出发，我们可以将Activity组件划分为两种类型：一种是根Activity，另一种是子Activity。根Activity以快捷图标的形式显示在应用程序启动器中，它的启动过程就代表了一个Android应用程序的启动过程。子Activity由根Activity或者其他子Activity启动，它们有可能与启动它们的Activity运行在同一个进程中，也有可能运行在不同的进程中，这取决于它们的配置和启动参数。

Activity组件的启动方式分为显示和隐式两种。对于显式启动的Activity组件来说，我们必须事先知道用来实现它们的类的名称；而对于隐式启动的Activity组件来说，我们只需要知道它们的组件名称即可，而不需要知道它们是由哪一个类来实现的。无论是显式启动的Activity组件，还是隐式启动的Activity组件，它们的启动过程都是类似的，唯一的区别在于系统是根据类名还是组件名称来找到它们。但是从软件工程的角度来看，隐式启动Activity组件可以减少Android应用程序组件间的依赖，因此，本章主要分析Activity组件的隐式启动过程。

接下来，我们首先通过一个应用实例来说明一个Activity组件的启动过程，然后分析应用程序根Activity的启动过程，最后分析应用程序子Activity在进程内和进程外的启动过程。

7.1 Activity组件应用实例

在本节中，我们将开发一个名称为Activity的Android应用程序，它由三个Activity组件MainActivity、SubActivityInProcess和SubActivityInNewProcess组成，其中，MainActivity是根Activity，SubActivityInProcess和SubActivityInNewProcess是子Activity。SubActivityInProcess与MainActivity运行在同一个进程中，而SubActivityInNewProcess运行在一个独立的进程中。

应用程序Activity的目录结构如下：

```
~/Android/packages/experimental/Activity
----AndroidManifest.xml
----Android.mk
----src
    ----shy/luo/activity
        ----MainActivity.java
```

```
        ----SubActivityInProcess.java
        ----SubActivityInNewProcess.java
----res
    ----layout
        ----main.xml
        ----sub.xml
    ----values
        ----strings.xml
    ----drawable
        ----icon.png
```

它包含了一个源代码目录src、一个资源目录res、一个配置文件AndroidManifest.xml和一个编译脚本文件Android.mk。接下来，我们就分别介绍每一个文件的内容。

MainActivity.java

```
01 package shy.luo.activity;
02
03 import android.app.Activity;
04 import android.content.Intent;
05 import android.os.Bundle;
06 import android.util.Log;
07 import android.view.View;
08 import android.view.View.OnClickListener;
09 import android.widget.Button;
10
11 public class MainActivity extends Activity  implements OnClickListener {
12     private final static String LOG_TAG = "shy.luo.activity.MainActivity";
13
14     private Button startInProcessButton = null;
15     private Button startInNewProcessButton = null;
16
17     @Override
18     public void onCreate(Bundle savedInstanceState) {
19         super.onCreate(savedInstanceState);
20         setContentView(R.layout.main);
21
22         startInProcessButton = (Button)findViewById(R.id.button_start_in_process);
23         startInNewProcessButton = (Button)findViewById(R.id.button_start_in_new_process);
24
25         startInProcessButton.setOnClickListener(this);
26         startInNewProcessButton.setOnClickListener(this);
27
28         Log.i(LOG_TAG, "Main Activity Created.");
29     }
30
31     @Override
32     public void onClick(View v) {
33         if(v.equals(startInProcessButton)) {
34                 Intent intent = new Intent("shy.luo.activity.subactivity.in.process");
35                 startActivity(intent);
36         } else if(v.equals(startInNewProcessButton)) {
37                 Intent intent = new Intent("shy.luo.activity.subactivity.in.new.process");
38                 startActivity(intent);
39         }
40     }
41 }
```

MainActivity是应用程序Activity的根Activity组件，它的用户界面上有两个按钮，分别用来启动SubActivityInProcess和SubActivityInNewProcess这两个子Activity。SubActivityInProcess和SubActivityInNewProcess的组件名称分别被配置为 “shy.luo.activity.subactivity.in.process”和“shy.luo.activity.subactivity.in.new.process”，因此，第35行和第38行调用成员函数startActivity启动它们时，只需要分别指定这两个名称即可，而不

需要知道它们是由哪一个类来实现的。

SubActivityInProcess.java

```
01 package shy.luo.activity;
02
03 import android.app.Activity;
04 import android.os.Bundle;
05 import android.util.Log;
06 import android.view.View;
07 import android.view.View.OnClickListener;
08 import android.widget.Button;
09
10 public class SubActivityInProcess extends Activity implements OnClickListener {
11     private final static String LOG_TAG = "shy.luo.activity.SubActivityInProcess";
12
13     private Button finishButton = null;
14
15     @Override
16     public void onCreate(Bundle savedInstanceState) {
17         super.onCreate(savedInstanceState);
18         setContentView(R.layout.sub);
19
20         finishButton = (Button)findViewById(R.id.button_finish);
21         finishButton.setOnClickListener(this);
22
23         Log.i(LOG_TAG, "Sub Activity In Process Created.");
24     }
25
26     @Override
27     public void onClick(View v) {
28         if(v.equals(finishButton)) {
29             finish();
30         }
31     }
32 }
```

SubActivityInProcess是应用程序Activity 的一个子Activity组件，它是由MainActivity组件启动起来的，当我们点击它的用户界面上的一个finish按钮时，就可以返回到MainActivity组件中。

SubActivityInNewProcess.java

```
01 package shy.luo.activity;
02
03 import android.app.Activity;
04 import android.os.Bundle;
05 import android.util.Log;
06 import android.view.View;
07 import android.view.View.OnClickListener;
08 import android.widget.Button;
09
10 public class SubActivityInNewProcess extends Activity implements OnClickListener {
11     private final static String LOG_TAG = "shy.luo.activity.SubActivityInNewProcess";
12
13     private Button finishButton = null;
14
15     @Override
16     public void onCreate(Bundle savedInstanceState) {
17         super.onCreate(savedInstanceState);
18         setContentView(R.layout.sub);
19
20         finishButton = (Button)findViewById(R.id.button_finish);
21         finishButton.setOnClickListener(this);
```

```
22
23          Log.i(LOG_TAG, "Sub Activity In New Process Created.");
24      }
25
26      @Override
27      public void onClick(View v) {
28          if(v.equals(finishButton)) {
29              finish();
30          }
31      }
32  }
```

SubActivityInNewProcess是应用程序Activity的另外一个子Activity组件，它也是由MainActivity组件启动起来的，不过它是在一个新的进程中启动的，当我们点击它的用户界面的finish按钮时，同样可以返回到MainActivity组件中。

main.xml

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
03      android:orientation="vertical"
04      android:layout_width="fill_parent"
05      android:layout_height="fill_parent"
06      android:gravity="center">
07      <Button
08          android:id="@+id/button_start_in_process"
09          android:layout_width="wrap_content"
10          android:layout_height="wrap_content"
11          android:gravity="center"
12          android:text="@string/start_in_process" >
13      </Button>
14      <Button
15          android:id="@+id/button_start_in_new_process"
16          android:layout_width="wrap_content"
17          android:layout_height="wrap_content"
18          android:gravity="center"
19          android:text="@string/start_in_new_process" >
20      </Button>
21 </LinearLayout>
```

这是MainActivity组件的界面配置文件，它在屏幕中间显示两个按钮。

sub.xml

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
03      android:orientation="vertical"
04      android:layout_width="fill_parent"
05      android:layout_height="fill_parent"
06      android:gravity="center">
07      <Button
08          android:id="@+id/button_finish"
09          android:layout_width="wrap_content"
10          android:layout_height="wrap_content"
11          android:gravity="center"
12          android:text="@string/finish" >
13      </Button>
14 </LinearLayout>
```

这是SubActivityInProcess组件和SubActivityInNewProcess组件的界面配置文件，它在屏幕中间显示一个按钮。

strings.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3     <string name="app_name">Activity</string>
4     <string name="sub_activity">Sub Activity</string>
5     <string name="start_in_process">Start sub-activity in process</string>
6     <string name="start_in_new_process">Start sub-activity in new process</string>
7     <string name="finish">Finish activity</string>
8 </resources>
```

这是应用程序Activity的字符串资源文件，定义了在应用程序中用到的各个字符串。

icon.png

这是应用程序Activity的图标，可以根据需要来放置不同的图片文件。

AndroidManifest.xml

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
03       package="shy.luo.activity"
04       android:versionCode="1"
05       android:versionName="1.0">
06       <application android:icon="@drawable/icon"
07                    android:label="@string/app_name">
08             <activity android:name=".MainActivity"
09                   android:label="@string/app_name"
10                   android:process="shy.luo.activity.mainprocess">
11                   <intent-filter>
12                       <action android:name="android.intent.action.MAIN" />
13                       <category android:name="android.intent.category.LAUNCHER" />
14                   </intent-filter>
15             </activity>
16             <activity android:label="@string/sub_activity"
17                   android:name=".SubActivityInProcess"
18                   android:process="shy.luo.activity.mainprocess">
19                   <intent-filter>
20                       <action android:name="shy.luo.activity.subactivity.in.process"/>
21                       <category android:name="android.intent.category.DEFAULT"/>
22                   </intent-filter>
23             </activity>
24             <activity android:label="@string/sub_activity"
25                   android:name=".SubActivityInNewProcess"
26                   android:process="shy.luo.activity.newprocess">
27                   <intent-filter>
28                       <action android:name="shy.luo.activity.subactivity.in.new.process"/>
29                       <category android:name="android.intent.category.DEFAULT"/>
30                   </intent-filter>
31             </activity>
32     </application>
33 </manifest>
```

这是应用程序Activity的配置文件。由于在程序中使用到了三个Activity组件MainActivity、SubActivityInProcess和SubActivityInNewProcess，因此，我们就需要在这里对它们进行配置。

注意

我们将MainActivity组件的Action名称和Category名称分别设置为“android.intent.action.MAIN”和“android.intent.category.LAUNCHER”，使得它可以作为应用程序Activity的根Activity组件。同时，我们将SubActivityInProcess组件和SubActivityInNewProcess组件的Action名称分别设置为自定义

值“shy.luo.activity.subactivity.in.process”和“shy.luo.activity.subactivity.in.new.process”，因此，可以通过这两个名称来隐式地启动它们。此外，我们还将MainActivity组件和SubActivityInProcess组件的android:process属性设置为“shy.luo.activity.mainprocess”，以便它们可以运行在同一个进程中；而将SubActivityInNewProcess组件的android:process属性设置为“shy.luo.activity.newprocess”，以便它可以运行在另外一个进程中。

Android.mk

```
1 LOCAL_PATH:= $(call my-dir)
2 include $(CLEAR_VARS)
3 LOCAL_MODULE_TAGS := optional
4 LOCAL_SRC_FILES := $(call all-subdir-java-files)
5 LOCAL_PACKAGE_NAME := Activity
6 include $(BUILD_PACKAGE)
```

这是应用程序Activity的编译脚本文件。

上述文件都准备好之后，我们就可以对应用程序Activity进行编译和打包了。

```
USER@MACHINE:~/Android$ mmm ./packages/experimental/Activity/
USER@MACHINE:~/Android$ make snod
```

编译成功之后，就可以在out/target/product/gerneric/system/app目录下看到应用程序文件Activity.apk了；打包成功之后，应用程序Activity就包含在out/target/product/gerneric目录下的Android系统镜像文件system.img中了。

最后，我们使用新得到的系统镜像文件system.img来启动Android模拟器。

```
USER@MACHINE:~/Android$ emulator
```

Android模拟器运行起来之后，我们就可以在应用程序启动器中启动应用程序Activity了，即将它的根Activity组件MainActivity启动起来，它的界面如图7–1所示。

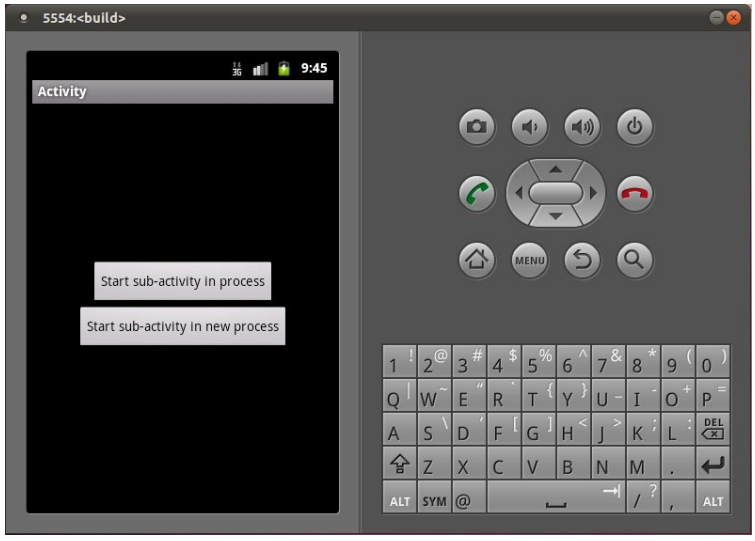


图7–1　应用程序Activity的MainActivity组件界面

点击MainActivity组件界面上的两个按钮，就可以分别将SubActivityInProcess组件和SubActivityInNewProcess组件启动起来，它们的界面如图7–2所示。

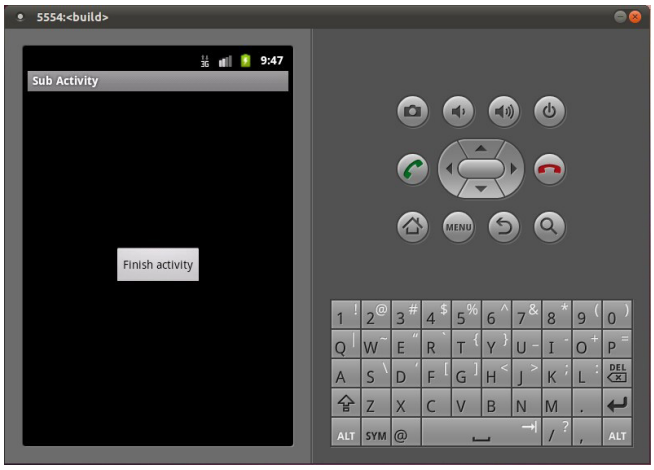


图7-2　应用程序Activity的SubActivityInProcess和SubActivityInNewProcess组件界面

接下来，我们就以MainActivity、SubActivityInProcess和SubActivityInNewProcess这三个组件的启动过程为例，深入分析Activity组件的启动过程。

7.2　根Activity组件的启动过程

本节将以应用程序Activity的MainActivity组件的启动过程为例，来说明Android应用程序的根Activity组件的启动过程，即Android应用程序的启动过程。

由于根Activity组件代表了一个Android应用程序，因此，它一般是在一个新的进程中启动起来的。在前面的7.1节中，当MainActivity组件被应用程序启动器中的Launcher组件启动起来之后，我们就可以通过adb shell dumpsys activity命令查看到它的信息，如下所示。

```
USER@MACHINE:~/Android$ adb shell dumpsys activity
  ......
  Running activities (most recent first):
    TaskRecord{4060ad30 #3 A shy.luo.activity}
      Run #1: HistoryRecord{4063fb08 shy.luo.activity/.MainActivity}
    TaskRecord{4064b4d8 #2 A com.android.launcher}
      Run #0: HistoryRecord{40551648 com.android.launcher/com.android.launcher2.Launcher}
  ......
  Running processes (most recent first):
    ......
    Proc # 2: adj=fore /F 4060ab58 377:shy.luo.activity.mainprocess/10036 (top-activity)
    ......
  ......
```

在Android系统中， 所有的Activity组件都保存在一个堆栈中，其中，后面启动的Activity组件位于前面启动的Activity组件的上面。用户在执行一个操作时，通常需要与一系列相关的Activity组件进行交互，这些相关的Activity组件在系统的Activity堆栈中用一个任务（Task）来描述[1]。

从Running activities小节的信息可以看出，系统的Activity组件堆栈有TaskRecord#2和TaksRecord#3两个任务。其中，任务TaskRecord#2包含了一个Launcher组件，表示我们打开了应用程序Launcher；而任务TaksRecord#3包含了一个MainActivity组件，表示我们打开了应用程序Activity。由于应用程序Activity是在应用程序Launcher中启动的，因此，任务TaksRecord#3就位于任务TaskRecord#2的上面。

1　Activity组件和任务（Task）的关系，可以参考官方文档：http://developer.android.com/guide/topics/fundamentals/tasks-and-back-stack.html。

由于ManActivity组件的android:process属性被设置为“shy.luo.activity.mainporcess”，因此，Activity管理服务ActivityManagerService在启动MainActivity组件时，就会发现系统中并不存在一个名称为“shy.luo.activity.mainporcess”的应用程序进程，这时候它就会先创建这个应用程序进程，然后再在里面将MainActivity组件启动起来。我们可以从Running processes小节的信息来证实这一点。

MainActivity组件是由Launcher组件来启动的，而Launcher组件又是通过Activity管理服务ActivityManagerService来启动MainActivity组件的[1]。由于MainActivity组件、Launcher组件和ActivityManagerService是分别运行在不同的进程中的，因此，MainActivity组件的启动过程就涉及到了三个进程。这三个进程是通过Binder进程间通信机制来完成MainActivity组件的启动过程的。

Launcher组件启动MainActivity组件的过程如下所示。

（1）Launcher组件向ActivityManagerService发送一个启动MainActivity组件的进程间通信请求。

（2）ActivityManagerService首先将要启动的MainActivity组件的信息保存下来，然后再向Launcher组件发送一个进入中止状态的进程间通信请求。

（3）Launcher组件进入到中止状态之后，就会向ActivityManagerService发送一个已进入中止状态的进程间通信请求，以便ActivityManagerService可以继续执行启动MainActivity组件的操作。

（4）ActivityManagerService发现用来运行MainActivity组件的应用程序进程不存在，因此，它就会先启动一个新的应用程序进程。

（5）新的应用程序进程启动完成之后，就会向ActivityManagerService发送一个启动完成的进程间通信请求，以便ActivityManagerService可以继续执行启动MainActivity组件的操作。

（6）ActivityManagerService将第2步保存下来的MainActivity组件的信息发送给第4步创建的应用程序进程，以便它可以将MainActivity组件启动起来。

这个过程一共分为35个步骤，接下来我们就详细分析每一个步骤。

第1步到第5步如图7-3所示。

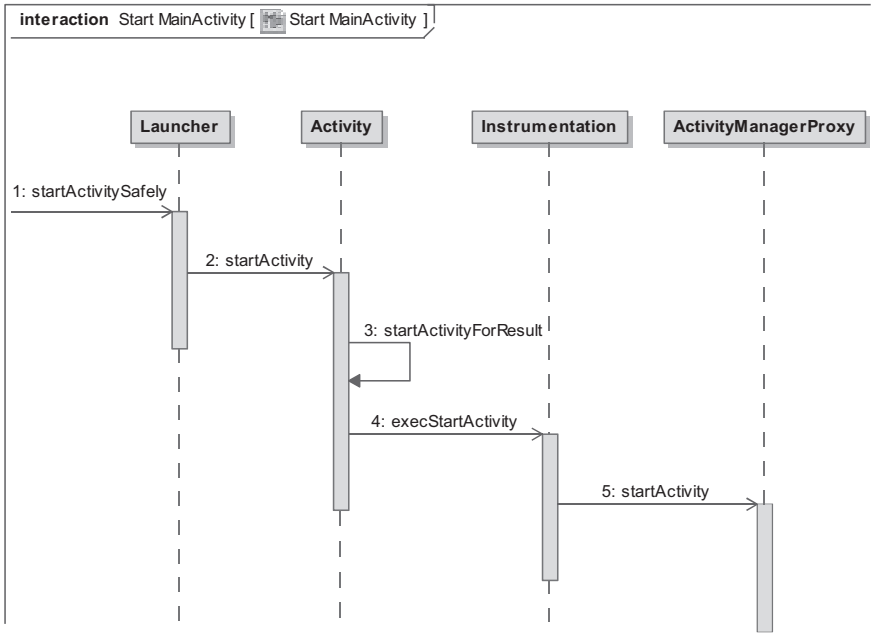


图7-3　MainActivity组件的启动过程——Step 1至Step 5

1　ActivityManagerService是一个系统关键服务，它运行在系统进程System中，负责启动和调度应用程序组件，它的启动过程可以参考第11章——Zygote和System进程的启动过程。

Step 1：Launcher.startActivitySafely

packages/apps/Launcher2/src/com/android/launcher2/Launcher.java

```
01 public final class Launcher extends Activity
02     implements View.OnClickListener, OnLongClickListener, LauncherModel.Callbacks, AllAppsView.Watcher {
03     ......
04
05     void startActivitySafely(Intent intent, Object tag) {
06         intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
07         try {
08             startActivity(intent);
09         } catch (ActivityNotFoundException e) {
10             ......
11         } catch (SecurityException e) {
12             ......
13         }
14     }
15
16     ......
17 }
```

当我们在应用程序启动器Launcher的界面上点击一个应用程序的快捷图标时，Launcher组件的成员函数startActivitySafely就会被调用来启动这个应用程序的根Activity，其中，要启动的根Activity的信息包含在参数intent中。

例如，当应用程序Activity的MainActivity组件被Launcher组件启动时，参数intent所包含的Activity组件信息就如下所示。

```
1 action = "android.intent.action.MAIN"
2 category = "android.intent.category.LAUNCHER"
3 cmp = "shy.luo.activity.MainActivity"
```

第1行和第2行表示要启动的Activity组件的Action名称和Category名称分别为“android.intent.action.MAIN”和“android.intent.category.LAUNCHER”，而第3行表示这个Activity组件是由shy.luo.activity.MainActivity类来实现的。

Laucher组件是如何获得这些信息的呢？系统在启动时，会启动一个Package管理服务PackageManagerService，并且通过它来安装系统中的应用程序。PackageManagerService在安装一个应用程序的过程中，会对它的配置文件AndroidManifest.xml进行解析，从而得到它里面的组件信息。系统在启动完成之后，就会将Launcher组件启动起来。Laucher组件在启动过程中，会向PackageManagerService查询所有Action名称等于“Intent.ACTION_MAIN”，并且Category名称等于“Intent.CATEGORY_LAUNCHER”的Activity组件，最后为每一个Activity组件创建一个快捷图标，并且将它们的信息与各自的快捷图标关联起来，以便用户点击它们时可以将对应的Activity组件启动起来[1]。

回到Launcher组件的成员函数startActivitySafely中，在第8行调用父类Activity的成员函数startActivity来启动参数intent所描述的Activity组件之前，第6行还会将这个Activity组件的启动标志位Intent.FLAG_ACTIVITY_NEW_TASK设置为1，以便它可以在一个新的任务中启动。

Step 2：Activity.startActivity

frameworks/base/core/java/android/app/Activity.java

```
01 public class Activity extends ContextThemeWrapper
02         implements LayoutInflater.Factory,
03         Window.Callback, KeyEvent.Callback,
```

1 Android应用程序的安装和显示过程可以参考第16章——Android应用程序的安装和显示过程。

```
04          OnCreateContextMenuListener, ComponentCallbacks {
05
06      ......
07
08      @Override
09      public void startActivity(Intent intent) {
10          startActivityForResult(intent, -1);
11      }
12
13      ......
14
15  }
```

第10行调用成员函数startActivityForResult来启动参数intent所描述的Activity组件，其中，第二个参数设置为-1，表示Launcher组件不需要知道即将启动的Actvity组件的执行结果。

Step 3：Activity.startActivityForResult

frameworks/base/core/java/android/app/Activity.java

```
01 public class Activity extends ContextThemeWrapper
02          implements LayoutInflater.Factory,
03          Window.Callback, KeyEvent.Callback,
04          OnCreateContextMenuListener, ComponentCallbacks {
05      ......
06
07      public void startActivityForResult(Intent intent, int requestCode) {
08          if (mParent == null) {
09              Instrumentation.ActivityResult ar =
10                  mInstrumentation.execStartActivity(
11                  this, mMainThread.getApplicationThread(), mToken, this,
12                  intent, requestCode);
13              ......
14          } else {
15              ......
16          }
17
18      ......
19  }
```

Activity类的成员变量mInstrumentation的类型为Intrumentation，它用来监控应用程序和系统之间的交互操作。由于启动Activity组件是应用程序与系统之间的一个交互操作，因此，第10行到第12行代码就调用它的成员函数execStartActivity来代为执行启动Activity组件的操作，以便它可以监控这个交互过程。

Activity类的成员变量mMainThread的类型为ActivityThread，用来描述一个应用程序进程。系统每当启动一个应用程序进程时，都会在它里面加载一个ActivityThread类实例，并且会将这个ActivityThread类实例保存在每一个在该进程中启动的Activity组件的父类Activity的成员变量mMainThread中。ActivityThread类的成员函数getApplicationThread用来获取它内部的一个类型为ApplicationThread的Binder本地对象。第11行将Launcher组件所运行在的应用程序进程的ApplicationThread对象作为参数传递给成员变量mInstrumentation的成员函数execStartActivity，以便可以将它传递给ActivityManagerService，这样ActivityManagerService接下来就可以通过它来通知Launcher组件进入Paused状态了。

Activity类的成员变量mToken的类型为IBinder，它是一个Binder代理对象，指向了ActivityManagerService中一个类型为ActivityRecord的Binder本地对象。每一个已经启动的Activity组件在ActivityManagerService中都有一个对应的ActivityRecord对象，用来维护对应的Activity组件的运行状态以及信息。第11行将Launcher组件的成员变量mToken作为参数传递给成员变量mInstrumentation的成员

函数execStartActivity，以便可以将它传递给ActivityManagerService，这样ActivityManagerService接下来就可以获得Launcher组件的详细信息了。

Step 4：Instrumentation.execStartActivity

frameworks/base/core/java/android/app/Instrumentation.java

```
01 public class Instrumentation {
02     ......
03
04     public ActivityResult execStartActivity(
05         Context who, IBinder contextThread, IBinder token, Activity target,
06         Intent intent, int requestCode) {
07         ......
08
09         try {
10             int result = ActivityManagerNative.getDefault()
11                 .startActivity(whoThread, intent,
12                 intent.resolveTypeIfNeeded(who.getContentResolver()),
13                 null, 0, token, target != null ? target.mEmbeddedID : null,
14                 requestCode, false, false);
15             ......
16         } catch (RemoteException e) {
17         }
18         return null;
19     }
20
21     ......
22 }
```

第10行到第14行代码首先调用ActivityManagerNative类的静态成员函数getDefault来获得ActivityManagerService的一个代理对象，接着再调用它的成员函数startActivity来通知ActivityManagerService将一个Activity组件启动起来。

ActivityManagerNative类的静态成员函数getDefault的实现如下所示。

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
01 public abstract class ActivityManagerNative extends Binder implements IActivityManager
02 {
03     ......
04
05     static public IActivityManager asInterface(IBinder obj)
06     {
07         if (obj == null) {
08             return null;
09         }
10         IActivityManager in =
11             (IActivityManager)obj.queryLocalInterface(descriptor);
12         if (in != null) {
13             return in;
14         }
15
16         return new ActivityManagerProxy(obj);
17     }
18
19     ......
20
21     static public IActivityManager getDefault()
22     {
23         if (gDefault != null) {
24             ......
25             return gDefault;
26         }
```

```
27          IBinder b = ServiceManager.getService("activity");
28          ......
29          gDefault = asInterface(b);
30          ......
31          return gDefault;
32      }
33
34      ......
35
36      private static IActivityManager gDefault;
37  }
```

第一次调用ActivityManagerNative类的静态成员函数getDefault时，它便会通过Service Manager来获得一个名称为“activity”的Java服务代理对象，即获得一个引用了ActivityManagerService的代理对象。接着第29行调用静态成员函数asInterface将这个代理对象封装成一个类型为ActivityManagerProxy的代理对象，最后将它保存在静态成员变量gDefault中。这样，以后再调用ActivityManagerNative类的静态成员函数getDefault时，就可以直接获得ActivityManagerService的一个代理对象了。

回到Instrumentation类的成员函数execStartActivity中，接下来就调用ActivityManagerProxy类的成员函数startActivity通知ActivityManagerService启动一个Activity组件。

Step 5：ActivityManagerProxy.startActivity

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
01 class ActivityManagerProxy implements IActivityManager
02 {
03     ......
04
05     public int startActivity(IApplicationThread caller, Intent intent,
06             String resolvedType, Uri[] grantedUriPermissions, int grantedMode,
07             IBinder resultTo, String resultWho,
08             int requestCode, boolean onlyIfNeeded,
09             boolean debug) throws RemoteException {
10         Parcel data = Parcel.obtain();
11         Parcel reply = Parcel.obtain();
12         data.writeInterfaceToken(IActivityManager.descriptor);
13         data.writeStrongBinder(caller != null ? caller.asBinder() : null);
14         intent.writeToParcel(data, 0);
15         data.writeString(resolvedType);
16         data.writeTypedArray(grantedUriPermissions, 0);
17         data.writeInt(grantedMode);
18         data.writeStrongBinder(resultTo);
19         data.writeString(resultWho);
20         data.writeInt(requestCode);
21         data.writeInt(onlyIfNeeded ? 1 : 0);
22         data.writeInt(debug ? 1 : 0);
23         mRemote.transact(START_ACTIVITY_TRANSACTION, data, reply, 0);
24         reply.readException();
25         int result = reply.readInt();
26         reply.recycle();
27         data.recycle();
28         return result;
29     }
30
31     ......
32 }
```

第12行到第22行代码将从前面传进来的参数写入到Parcel对象data中，接着再通过ActivityManagerProxy类内部的一个Binder代理对象mRemote向ActivityManagerService发送一个类型为START_ACTIVITY_TRANSACTION的进程间通信请求。

传递给ActivityManagerService的参数比较多，不过我们需要重点关注的参数有三个，它们分别是caller、intent和resultTo。其中，参数caller指向Launcher组件所运行在的应用程序进程的ApplicationThread对象；参数intent包含了即将要启动的MainActivity组件的信息；参数resultTo指向ActivityManagerService内部的一个ActivityRecord对象，它里面保存了Launcher组件的详细信息。

以上5步都是在应用程序Launcher中执行的，接下来第6步到第12步是在ActivityManagerService中执行的，主要用来处理Launcher组件发出的类型为START_ACTIVITY_TRANSACTION的进程间通信请求，如图7-4所示。

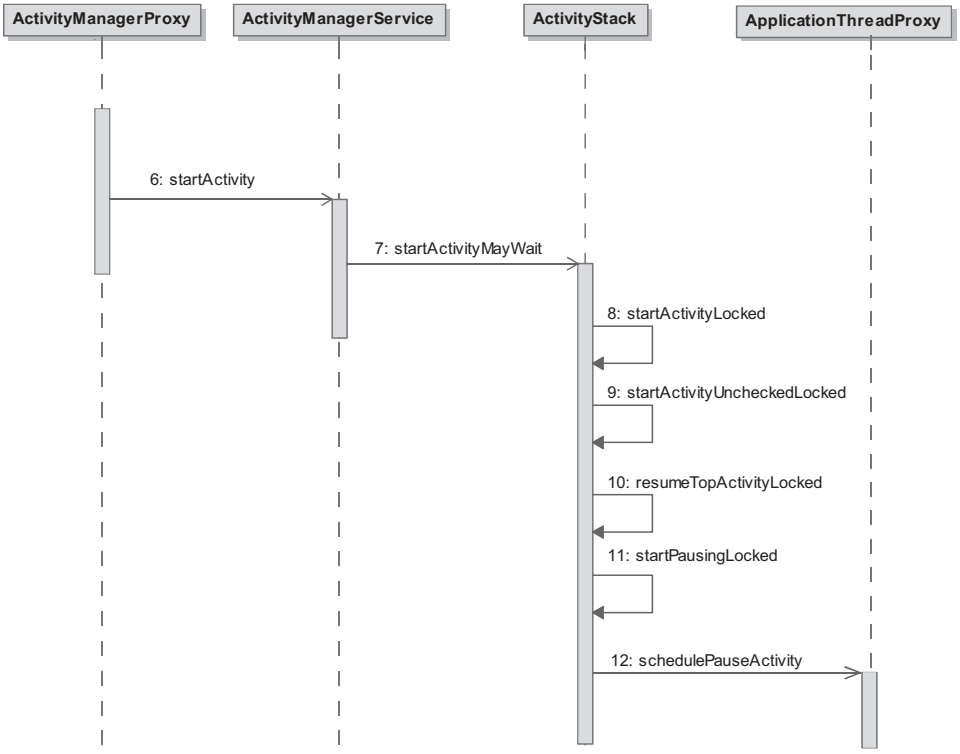


图7-4　MainActivity组件的启动过程——Step 6至Step 12

Step 6：ActivityManagerService.startActivity

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     public final int startActivity(IApplicationThread caller,
06             Intent intent, String resolvedType, Uri[] grantedUriPermissions,
07             int grantedMode, IBinder resultTo,
08             String resultWho, int requestCode, boolean onlyIfNeeded,
09             boolean debug) {
10         return mMainStack.startActivityMayWait(caller, intent, resolvedType,
11             grantedUriPermissions, grantedMode, resultTo, resultWho,
12             requestCode, onlyIfNeeded, debug, null, null);
13     }
14
15     ......
16 }
```

ActivityManagerService类的成员函数startActivity 用来处理类型为START_ACTIVITY_TRANSACTION的进程间通信请求。

ActivityManagerService类有一个类型为ActivityStack的成员变量mMainStack，用来描述一个Activity组件堆栈，第10行到第12行代码调用它的成员函数startActivityMayWait 来进一步处理类型为START_ACTIVITY_TRANSACTION的进程间通信请求，即执行一个启动Activity组件的操作。

Step 7：ActivityStack.startActivityMayWait

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02     ......
03
04     final int startActivityMayWait(IApplicationThread caller,
05             Intent intent, String resolvedType, Uri[] grantedUriPermissions,
06             int grantedMode, IBinder resultTo,
07             String resultWho, int requestCode, boolean onlyIfNeeded,
08             boolean debug, WaitResult outResult, Configuration config) {
09         ......
10
11         boolean componentSpecified = intent.getComponent() != null;
12
13         // Don't modify the client's object!
14         intent = new Intent(intent);
15
16         // Collect information about the target of the Intent.
17         ActivityInfo aInfo;
18         try {
19             ResolveInfo rInfo =
20                 AppGlobals.getPackageManager().resolveIntent(
21                 intent, resolvedType,
22                 PackageManager.MATCH_DEFAULT_ONLY
23                 | ActivityManagerService.STOCK_PM_FLAGS);
24             aInfo = rInfo != null ? rInfo.activityInfo : null;
25         } catch (RemoteException e) {
26             ......
27         }
28
29         ......
30
31         synchronized (mService) {
32             ......
33
34             int res = startActivityLocked(caller, intent, resolvedType,
35                 grantedUriPermissions, grantedMode, aInfo,
36                 resultTo, resultWho, requestCode, callingPid, callingUid,
37                 onlyIfNeeded, componentSpecified);
38
39             ......
40
41             return res;
42         }
43
44     }
45
46     ......
47 }
```

第19行到第23行代码到Pacakge管理服务PacakgeManagerService中去解析参数intent的内容，以便可以获得即将启动的Activity组件的更多信息；接着第24行将这些信息保存在ActivityInfo对象aInfo中，最后第34行到第37行代码调用成员函数startActivityLocked来继续执行启动Activity组件的工作。

Step 8：ActivityStack.startActivityLocked

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02     ......
03
04     final int startActivityLocked(IApplicationThread caller,
05             Intent intent, String resolvedType,
06             Uri[] grantedUriPermissions,
07             int grantedMode, ActivityInfo aInfo, IBinder resultTo,
08                 String resultWho, int requestCode,
09             int callingPid, int callingUid, boolean onlyIfNeeded,
10             boolean componentSpecified) {
11             int err = START_SUCCESS;
12
13         ProcessRecord callerApp = null;
14         if (caller != null) {
15             callerApp = mService.getRecordForAppLocked(caller);
16             if (callerApp != null) {
17                 callingPid = callerApp.pid;
18                 callingUid = callerApp.info.uid;
19             } else {
20                 ......
21             }
22         }
23
24         ......
25
26         ActivityRecord sourceRecord = null;
27         ......
28         if (resultTo != null) {
29             int index = indexOfTokenLocked(resultTo);
30             ......
31
32             if (index >= 0) {
33                 sourceRecord = (ActivityRecord)mHistory.get(index);
34                 ......
35             }
36         }
37
38         ......
39
40         ActivityRecord r = new ActivityRecord(mService, this, callerApp, callingUid,
41             intent, resolvedType, aInfo, mService.mConfiguration,
42             resultRecord, resultWho, requestCode, componentSpecified);
43
44         ......
45
46         return startActivityUncheckedLocked(r, sourceRecord,
47             grantedUriPermissions, grantedMode, onlyIfNeeded, true);
48     }
49
50     ......
51 }
```

在ActivityManagerService中，每一个应用程序进程都使用一个ProcessRecord对象来描述，并且保存在ActivityManagerService内部。ActivityStack类的成员变量mService指向了ActivityManagerService，第15行通过调用它的成员函数getRecordForAppLocked来获得与参数caller对应的一个ProcessRecord对象callerApp。前面提到，参数caller指向的是Launcher组件所运行在的应用程序进程的一个ApplicationThread对象，因此，第15行得到的ProcessRecord对象callerApp实际上就指向了Launcher组件

所运行在的应用程序进程。接着第17行和第18行得到这个应用程序进程的PID以及UID，保存在参数callingPid和callingUid中。

ActivityStack类的成员变量mHistory用来描述系统的Activity组件堆栈。在这个堆栈中，每一个已经启动的Activity组件都使用一个ActivityRecord对象来描述。前面提到，参数resultTo指向的是Launcher组件在ActivityManagerService中的一个ActivityRecord对象，因此，第29行到第35行代码实际上就是在Activity组件堆栈mHistory中找到用来描述Launcher组件的一个ActivityRecord对象，保存在变量sourceRecord中。

第40行到第42行代码创建一个ActivityRecord对象r来描述即将启动的Activity组件，即MainActivity组件。

现在，ActivityStack类就得到了请求ActivityManagerService执行启动Activity组件操作的源Activity组件，以及要启动的目标Activity组件的信息了，它们分别保存在ActivityRecord对象sourceRecord和r中。最后，第46行就以它们为参数来调用成员函数startActivityUncheckedLocked进一步执行启动目标Activity组件的操作。

Step 9：ActivityStack.startActivityUncheckedLocked

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02     ......
03
04     final int startActivityUncheckedLocked(ActivityRecord r,
05         ActivityRecord sourceRecord, Uri[] grantedUriPermissions,
06         int grantedMode, boolean onlyIfNeeded, boolean doResume) {
07         final Intent intent = r.intent;
08         final int callingUid = r.launchedFromUid;
09
10         int launchFlags = intent.getFlags();
11
12         // We'll invoke onUserLeaving before onPause only if the launching
13         // activity did not explicitly state that this is an automated launch.
14         mUserLeaving = (launchFlags&Intent.FLAG_ACTIVITY_NO_USER_ACTION) == 0;
15
16         ......
17
18         boolean addingToTask = false;
19
20         ......
21
22         boolean newTask = false;
23
24         // Should this be considered a new task?
25         if (r.resultTo == null && !addingToTask
26             && (launchFlags&Intent.FLAG_ACTIVITY_NEW_TASK) != 0) {
27                 // todo: should do better management of integers.
28                 mService.mCurTask++;
29                 if (mService.mCurTask <= 0) {
30                     mService.mCurTask = 1;
31                 }
32                 r.task = new TaskRecord(mService.mCurTask, r.info, intent,
33                     (r.info.flags&ActivityInfo.FLAG_CLEAR_TASK_ON_LAUNCH) != 0);
34                 ......
35                 newTask = true;
36                 if (mMainStack) {
37                     mService.addRecentTaskLocked(r.task);
38                 }
39
40         }
```

```
41
42          ......
43
44          startActivityLocked(r, newTask, doResume);
45          return START_SUCCESS;
46      }
47
48      ......
49  }
```

第10行得到目标Activity组件的启动标志位，保存在变量launchFlags中。从前面的Step 1可以知道，在变量launchFlags中，只有Intent.FLAG_ACTIVITY_NEW_TASK位被设置为1，其他位均等于0。

第14行检查变量launchFlags的Intent.FLAG_ACTIVITY_NO_USER_ACTION位是否等于1。如果等于1，那么就表示目标Activity组件不是由用户手动启动的。如果目标Activity组件是由用户手动启动的，那么用来启动它的源Activity组件就会获得一个用户离开事件通知。由于目标Activity组件是用户在应用程序启动器的界面上点击启动的，即变量launchFlags的Intent.FLAG_ACTIVITY_NO_USER_ACTION位等于0；因此，第14行得到成员变量mUserLeaving的值就等于true，表示后面要向源Activity组件发送一个用户离开事件通知。

在默认情况下，目标Activity组件是与源Activity组件运行在同一个任务中的。然而，如果源Activity组件将目标Activity组件的启动标志值的FLAG_ACTIVITY_NEW_TASK位设置为1，并且源Activity组件不需要知道目标Activity组件的运行结果，那么ActivityManagerService就会将目标Activity组件运行在另外一个不同的任务中。这个不同的任务可能是一个新创建的任务，也可能是一个已经存在的任务。Activity组件有一个android:taskAffinity属性，用来描述它的一个专属任务。当ActivityManagerService决定要将目标Activity组件运行在一个不同的任务中时，ActivityManagerService就会检查目标Activity组件的专属任务是否已经存在。如果已经存在，那么ActivityManagerService就会直接将目标Activity组件添加到它里面去运行；否则，就会先创建这个专属任务，然后再将目标Activity组件添加到它里面去运行。

第18行将变量addingToTask的值初始化为false，表示要为目标Activity组件创建一个专属任务，事实上函数接下来会检查这个专属任务是否已经存在。如果已经存在，那么就会将变量addingToTask的值设置为true。为了描述方便，我们忽略了检查目标Activity组件的专属任务是否已经存在的代码，并且假设不存在，即最终变量addingToTask的值保持为false。

从前面的Step 2可以知道，源Activity组件是不需要知道目标Activity组件的执行结果的，即参数r的成员变量resultTo的值等于null。综合前面得到的变量launchFlags和addingToTask的值，我们就可以知道第25行的if语句为true；因此，第32行和第33行代码就会为目标Activity组件创建一个专属任务，并且将它保存在参数r的成员变量task中。接着第35行继续将变量newTask的值设置为true，并且第37行将新创建的专属任务交给ActivityManagerService来管理。

最后，第44行调用另外一个重载版本的成员函数startActivityLocked来执行启动目标Activity组件的操作，它的实现如下所示。

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02     ......
03
04     private final void startActivityLocked(ActivityRecord r, boolean newTask,
05             boolean doResume) {
06         final int NH = mHistory.size();
07
08         int addPos = -1;
09         ......
```

```
10
11          // Place a new activity at top of stack, so it is next to interact
12          // with the user.
13          if (addPos < 0) {
14              addPos = NH;
15          }
16          ......
17
18          // Slot the activity into the history stack and proceed
19          mHistory.add(addPos, r);
20          ......
21
22          if (doResume) {
23              resumeTopActivityLocked(null);
24          }
25      }
26
27      ......
28  }
```

第6行到第19行代码将目标Activity组件保存到ActivityStack类的成员变量mHistory所描述的Activity组件堆栈中。当目标Activity组件是在一个新创建的任务中启动时，即参数newTask的值等于true时，ActivityStack就需要将它放在Activity组件堆栈的最上面，即将变量addPos的值设置为Activity组件堆栈的当前大小NH，以便接下来可以将它激活。

从前面的Step 8可以知道，传进来的参数doResume的值等于true；因此，接下来第23行就会调用成员函数resumeTopActivityLocked将Activity组件堆栈顶端的Activity组件激活，这个Activity组件就正好是即将要启动的MainActivity组件，因为前面第19行正好将它放在了Activity组件堆栈的顶端。

Step 10：ActivityStack.resumeTopActivityLocked

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02      ......
03
04      final boolean resumeTopActivityLocked(ActivityRecord prev) {
05          // Find the first activity that is not finishing.
06          ActivityRecord next = topRunningActivityLocked(null);
07
08          // Remember how we'll process this pause/resume situation, and ensure
09          // that the state is reset however we wind up proceeding.
10          final boolean userLeaving = mUserLeaving;
11          mUserLeaving = false;
12          ......
13
14
15          // If the top activity is the resumed one, nothing to do.
16          if (mResumedActivity == next && next.state == ActivityState.RESUMED) {
17              ......
18              return false;
19          }
20
21          // If we are sleeping, and there is no resumed activity, and the top
22          // activity is paused, well that is the state we want.
23          if ((mService.mSleeping || mService.mShuttingDown)
24              && mLastPausedActivity == next && next.state == ActivityState.PAUSED) {
25              ......
26              return false;
27          }
28          ......
29
```

```
30          // If we are currently pausing an activity, then don't do anything
31          // until that is done.
32          if (mPausingActivity != null) {
33              ......
34              return false;
35          }
36          ......
37
38          // We need to start pausing the current activity so the top one
39          // can be resumed...
40          if (mResumedActivity != null) {
41              ......
42              startPausingLocked(userLeaving, false);
43              return true;
44          }
45
46          ......
47      }
48
49      ......
50  }
```

第6行调用成员函数topRunningActivityLocked来获得当前Activity组件堆栈最上面的一个不是处于结束状态的Activity组件。在前面的Step 9中，我们将MainActivity组件保存在当前Activity组件堆栈的顶端，并且它是正在等待启动的，即它不是处于结束状态的，因此，第6行得到的ActivityRecord对象next就指向了即将要启动的MainActivity组件。

第10行将ActivityStack类的成员变量mUserLeaving的值保存在变量userLeaving中，并且第11行再将它重置为false，因为接下来通过变量userLeaving的值就可以知道是否需要向源Activity组件发送一个用户离开事件通知了。

ActivityStack类有三个成员变量mResumedActivity、mLastPausedActivity和mPausingActivity，它们的类型均为ActivityRecord，分别用来描述系统当前激活的Activity组件、上一次被中止的Activity组件，以及正在被中止的Activity组件。

第16行的if语句检查即将要启动的Activity组件next是否等于当前被激活的Activity组件。如果等于，并且它的状态为Resumed，那么第18行就可以直接返回了，因为要启动的Activity组件本身就已经启动和激活了。

第23行的if语句检查即将要启动的Activity组件next是否等于上一次被中止了的Activity组件。如果等于，并且这时候系统正要进入关机或者睡眠状态，那么第26行也可以直接返回了，因为这时候将这个Activity组件启动起来是没有意义的。

第32行的if语句检查系统当前是否正在中止一个Activity组件。如果是，那么就要等到它中止完成之后，再启动Activity组件next，因此，第34行也直接返回了。

由于系统当前正在激活的Activity组件是Launcher组件，即ActivityStack类的成员变量mResumedActivity指向了Launcher组件，因此，第40行的if语句就等于true。接下来第42行就会调用成员函数startPausingLocked来通知它进入Paused状态，以便它可以将焦点让给即将要启动的MainActivity组件。

Step 11：ActivityStack.startPausingLocked

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02      ......
03
04      private final void startPausingLocked(boolean userLeaving, boolean uiSleeping) {
05          ......
06
07          ActivityRecord prev = mResumedActivity;
```

```
08          ......
09
10          mResumedActivity = null;
11          mPausingActivity = prev;
12          mLastPausedActivity = prev;
13          prev.state = ActivityState.PAUSING;
14          ......
15
16          if (prev.app != null && prev.app.thread != null) {
17              ......
18              try {
19                  ......
20                  prev.app.thread.schedulePauseActivity(prev, prev.finishing, userLeaving,
21                      prev.configChangeFlags);
22                  ......
23              } catch (Exception e) {
24                  ......
25              }
26          } else {
27              ......
28          }
29          ......
30
31          if (mPausingActivity != null) {
32             ......
33
34             // Schedule a pause timeout in case the app doesn't respond.
35             // We don't give it much time because this directly impacts the
36             // responsiveness seen by the user.
37             Message msg = mHandler.obtainMessage(PAUSE_TIMEOUT_MSG);
38             msg.obj = prev;
39             mHandler.sendMessageDelayed(msg, PAUSE_TIMEOUT);
40             ......
41         }
42         ......
43      }
44
45      ......
46  }
```

第7行到第13行代码主要是将ActivityRecord对象prev，以及ActivityStack类的成员变量mPausingActivity和mLastPausedActivity指向即将要进入Paused状态的Launcher组件，并且将成员变量mResumedActivity的值设置为null。

ActivityRecord类有一个成员变量app，它的类型为ProcessRecord，用来描述一个Activity组件所运行在的应用程序进程；而ProcessRecord类又有一个成员变量thread，它的类型为 ApplicationThreadProxy，用来描述一个Binder代理对象，引用的是一个类型为ApplicationThread的Binder本地对象。

由于ActivityRecord对象prev指向的是即将要进入Paused状态的Launcher组件，因此，第20行代码实际上是向Launcher组件所运行在的应用程序进程发送一个中止Launcher组件的通知，以便Launcher组件有机会执行一些数据保存操作。

Launcher组件处理完ActivityManagerService给它发送的中止通知之后，必须再向ActivityManagerService发送一个启动MainActivity组件的通知，以便ActivityManagerService可以将位于Activity组件堆栈顶端的MainActivity组件启动起来。但是，ActivityManagerService不能无限时地等待，因此，第37行到第39行代码就向ActivityManagerService所运行在的线程的消息队列发送一个类型为PAUSE_TIMEOUT_MSG的消息，并且指定这个消息在PAUSE_TIMEOUT毫秒之后处理[1]。如果Launcher组件不能在PAUSE_TIMEOUT毫秒内再向ActivityManagerService发送一个启动MainActivity组件的通知，那么

1　Android线程的消息处理机制可以参考第13章——Android应用程序的消息处理机制。

ActivityManagerService就会认为它没有响应了。

接下来，我们继续分析 ApplicationThreadProxy类的成员函数schedulePauseActivity的实现。

Step 12：ApplicationThreadProxy.schedulePauseActivity

frameworks/base/core/java/android/app/ApplicationThreadNative.java

```
01 class ApplicationThreadProxy implements IApplicationThread {
02     ......
03
04     public final void schedulePauseActivity(IBinder token, boolean finished,
05              boolean userLeaving, int configChanges) throws RemoteException {
06         Parcel data = Parcel.obtain();
07         data.writeInterfaceToken(IApplicationThread.descriptor);
08         data.writeStrongBinder(token);
09         data.writeInt(finished ? 1 : 0);
10         data.writeInt(userLeaving ? 1 :0);
11         data.writeInt(configChanges);
12         mRemote.transact(SCHEDULE_PAUSE_ACTIVITY_TRANSACTION, data, null,
13             IBinder.FLAG_ONEWAY);
14         data.recycle();
15     }
16
17     .....
18 }
```

第7行到第11行代码将从前面传进来的参数写入到Parcel对象data中，然后再通过ApplicationThreadProxy类内部的一个Binder代理对象mRemote向Launcher组件所在的应用程序进程发送一个类型为SCHEDULE_PAUSE_ACTIVITY_TRANSACTION的进程间通信请求。

注意

这个进程间通信请求是异步的，因此，ActivityManagerService将它发送出去之后，就马上返回了。

以上7步都是在ActivityManagerService中执行的，接下来第13步到第17步是在应用程序Launcher中执行的，主要用来处理ActivityManagerService发出的类型为SCHEDULE_PAUSE_ACTIVITY_TRANSACTION的进程间通信请求，如图7-5所示。

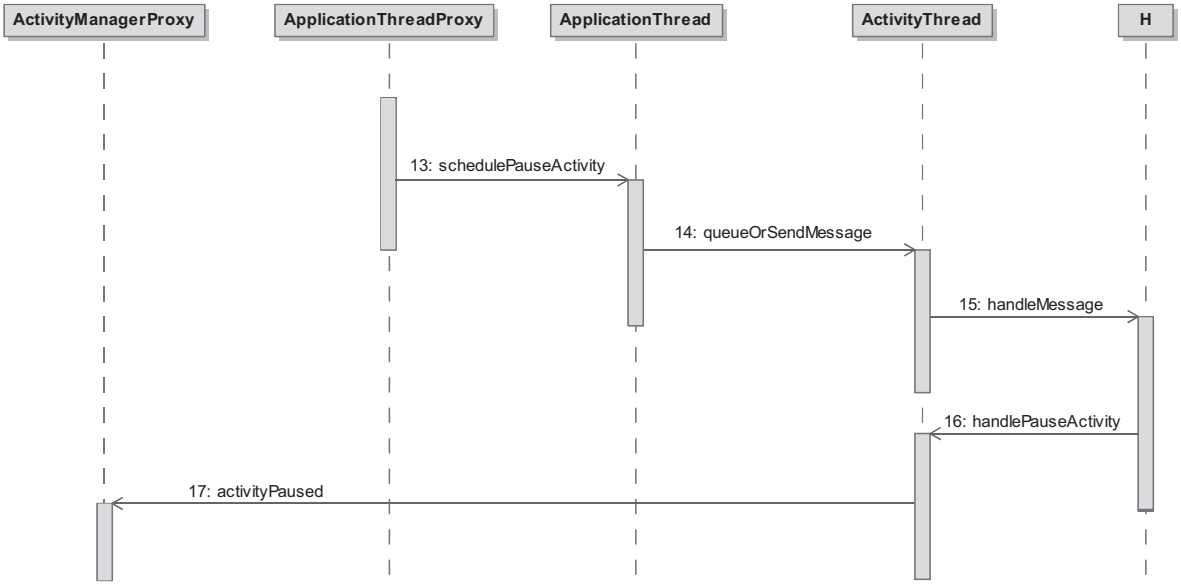


图7-5 MainActivity组件的启动过程——Step 13至Step 17

Step 13：ApplicationThread.schedulePauseActivity

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final class ApplicationThread extends ApplicationThreadNative {
05         ......
06
07         public final void schedulePauseActivity(IBinder token, boolean finished,
08                 boolean userLeaving, int configChanges) {
09             queueOrSendMessage(
10                 finished ? H.PAUSE_ACTIVITY_FINISHING : H.PAUSE_ACTIVITY,
11                 token,
12                 (userLeaving ? 1 : 0),
13                 configChanges);
14         }
15
16         ......
17     }
18
19     ......
20 }
```

ApplicationThread类的成员函数schedulePauseActivity 用来处理类型为SCHEDULE_PAUSE_ACTIVITY_TRANSACTION的进程间通信请求。

从前面传进来的参数finished和userLeaving分别为false和true，而参数token是一个Binder代理对象，指向了ActivityManagerService中与Launcher组件对应的一个ActivityRecord对象。

第9行到第13行代码调用外部类ActivityThread的成员函数queueOrSendMessage来向应用程序Launcher的主线程的消息队列发送一个类型为PAUSE_ACTIVITY的消息。

Step 14：ActivityThread.queueOrSendMessage

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     final H mH = new H();
05
06     ......
07
08     private final void queueOrSendMessage(int what, Object obj, int arg1) {
09         queueOrSendMessage(what, obj, arg1, 0);
10     }
11
12     private final void queueOrSendMessage(int what, Object obj, int arg1, int arg2) {
13         synchronized (this) {
14             ......
15             Message msg = Message.obtain();
16             msg.what = what;
17             msg.obj = obj;
18             msg.arg1 = arg1;
19             msg.arg2 = arg2;
20             mH.sendMessage(msg);
21         }
22     }
23
24     ......
25 }
```

ActivityThread类的成员变量mH是用来处理应用程序进程的主线程的消息的，它的类型为H，继承了Handler类。我们可以调用它的成员函数sendMessage来向应用程序进程的主线程的消息队列发送一个消息，这个消息最后由应用程序进程的主线程调用它的成员函数handleMessage来处理。

第15行到第19行代码将前面Step 13传进来的一个Binder代理对象封装成一个Message对象msg，接着第20行代码调用ActivityThread类的成员变量mH向应用程序Launcher的主线程的消息队列发送一个类型为PAUSE_ACTIVITY的消息，以便可以在应用程序Launcher的主线程中执行中止Launcher组件的操作。

注意

应用程序Launcher为什么不直接在当前线程中执行中止Launcher组件的操作呢？一方面是因为当前线程需要尽快返回到Binder线程池中去处理其他进程间通信请求；另一方面是因为在中止Launcher组件的过程中，可能会涉及用户界面相关的操作，因此就需要将它放在主线程中执行。

Step 15：H.handleMessage

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final class H extends Handler {
05         ......
06
07         public void handleMessage(Message msg) {
08             ......
09
10             switch (msg.what) {
11             ......
12             case PAUSE_ACTIVITY:
13                 handlePauseActivity((IBinder)msg.obj, false, msg.arg1 != 0, msg.arg2);
14                 ......
15                 break;
16             ......
17             }
18
19         ......
20     }
21
22     ......
23 }
```

第13行首先将Message对象msg的成员变量obj强制转换成一个IBinder接口，因为它指向的是一个Binder代理对象，接着再调用ActivityThread类的成员函数handlePauseActivity来处理这个类型为PAUSE_ACTIVITY的消息。

Step 16：ActivityThread.handlePauseActivity

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     final HashMap<IBinder, ActivityClientRecord> mActivities
05             = new HashMap<IBinder, ActivityClientRecord>();
06     ......
07
08     private final void handlePauseActivity(IBinder token, boolean finished,
```

```
09              boolean userLeaving, int configChanges) {
10
11          ActivityClientRecord r = mActivities.get(token);
12          if (r != null) {
13              //Slog.v(TAG, "userLeaving=" + userLeaving + " handling pause of " + r);
14              if (userLeaving) {
15                  performUserLeavingActivity(r);
16              }
17
18              r.activity.mConfigChangeFlags |= configChanges;
19              Bundle state = performPauseActivity(token, finished, true);
20
21              // Make sure any pending writes are now committed.
22              QueuedWork.waitToFinish();
23
24              // Tell the activity manager we have paused.
25              try {
26                  ActivityManagerNative.getDefault().activityPaused(token, state);
27              } catch (RemoteException ex) {
28              }
29          }
30      }
31
32      ......
33  }
```

在应用程序进程中启动的每一个Activity组件都使用一个ActivityClientRecord对象来描述，这些ActivityClientRecord对象对应于ActivityManagerService中的ActivityRecord对象，并且保存在ActivityThread类的成员变量mActivities中。

从前面的调用过程可以知道，参数token指向的是ActivityManagerService中与Launcher组件对应的一个ActivityRecord对象，因此，第11行就可以根据它的值在ActivityThread类的成员变量mActivities中找到一个用来描述Launcher组件的ActivityClientRecord对象。

获得了要中止的目标Activity组件之后，ActivityThread类的成员函数handlePauseActivity接下来做了三件事情。

第一件事情是调用成员函数performUserLeavingActivity向Launcher组件发送一个用户离开事件通知，即调用它的成员函数onUserLeaveHint。

第二件事情是调用成员函数performPauseActivity向Launcher组件发送一个中止事件通知，即调用它的成员函数onPause。

第三件事情是调用QueuedWork类的静态成员函数waitToFinish等待完成前面的一些数据写入操作，例如，将数据写入到磁盘的操作。由于现在Launcher组件即将要进入Paused状态了，因此就要保证它前面的所有数据写入操作都处理完成；否则，等到它重新进入Resumed状态时，就无法恢复之前所保存的一些状态数据。

执行完成这三件事情之后，ActivityThread类的成员函数handlePauseActivity就处理完ActivityManagerService给它发送的中止Launcher组件的进程间通信请求了。

接下来第26行调用ActivityManagerNative类的静态成员函数getDefault来获得ActivityManagerService的一个代理对象，然后再调用这个代理对象的成员函数activityPaused来通知ActivityManagerService，Launcher组件已经进入Paused状态了，因此，它就可以将MainActivity组件启动起来了。

Step 17：ActivityManagerProxy.activityPaused

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
01 class ActivityManagerProxy implements IActivityManager
02 {
03     ......
04
05     public void activityPaused(IBinder token, Bundle state) throws RemoteException
06     {
07         Parcel data = Parcel.obtain();
08         Parcel reply = Parcel.obtain();
09         data.writeInterfaceToken(IActivityManager.descriptor);
10         data.writeStrongBinder(token);
11         data.writeBundle(state);
12         mRemote.transact(ACTIVITY_PAUSED_TRANSACTION, data, reply, 0);
13         reply.readException();
14         data.recycle();
15         reply.recycle();
16     }
17
18     ......
19 }
```

第9行到第11行代码将从前面传进来的参数写入到Parcel对象data中，然后再通过ActivityManagerProxy类内部的一个Binder代理对象mRemote向ActivityManagerService发送一个类型为ACTIVITY_PAUSED_TRANSACTION的进程间通信请求。

以上5步都是在应用程序Launcher中执行的，接下来第18步到第23步是在ActivityManagerService中执行的，主要用来处理Launcher组件发出的类型为ACTIVITY_PAUSED_TRANSACTION的进程间通信请求，如图7-6所示。

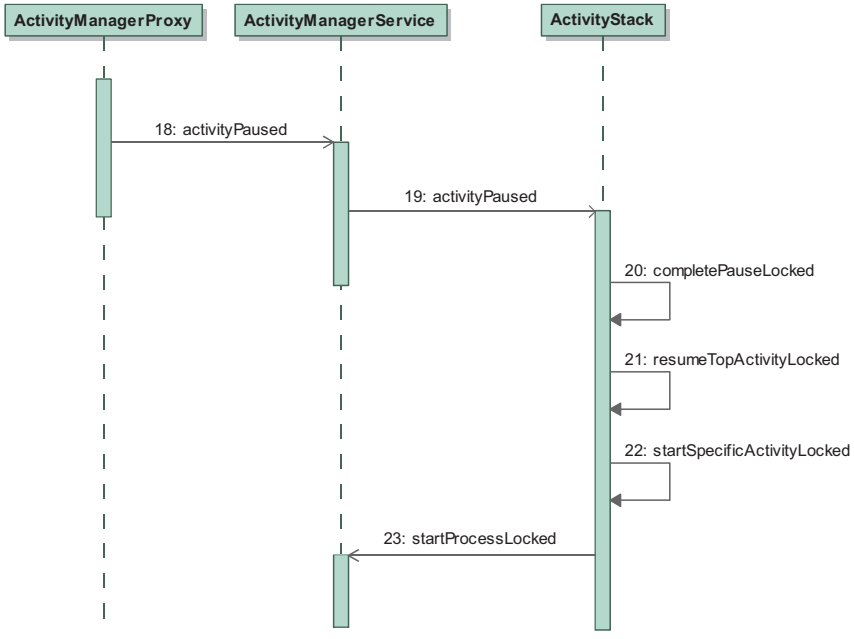


图7-6　MainActivity组件的启动过程——Step 18至Step 23

Step 18：ActivityManagerService.activityPaused

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02             implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
```

```
03      ......
04
05      public final void activityPaused(IBinder token, Bundle icicle) {
06          ......
07
08          mMainStack.activityPaused(token, icicle, false);
09
10          ......
11      }
12
13      ......
14  }
```

ActivityManagerService类的成员函数activityPaused用来处理类型为ACTIVITY_PAUSED_TRANSACTION的进程间通信请求。

注意

从前面传进来的参数token指向了ActivityManagerService中与Launcher组件对应的一个ActivityRecord对象。第8行调用ActivityManagerService的成员变量mMainStack 的成员函数activityPaused来继续执行启动MainActivity组件的操作。

Step 19：ActivityStack.activityPaused

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02      ......
03
04      final void activityPaused(IBinder token, Bundle icicle, boolean timeout) {
05          ......
06
07          ActivityRecord r = null;
08
09          synchronized (mService) {
10              int index = indexOfTokenLocked(token);
11              if (index >= 0) {
12                  r = (ActivityRecord)mHistory.get(index);
13                  if (!timeout) {
14                      r.icicle = icicle;
15                      r.haveState = true;
16                  }
17                  mHandler.removeMessages(PAUSE_TIMEOUT_MSG, r);
18                  if (mPausingActivity == r) {
19                      r.state = ActivityState.PAUSED;
20                      completePauseLocked();
21                  } else {
22                      ......
23                  }
24              }
25          }
26      }
27
28      ......
29  }
```

第10行到第12行代码首先根据参数token的值在Activity组件堆栈mHistory中找到与Launcher组件对应的ActivityRecord对象r。

第17行删除ActivityManagerService所运行在的线程的消息队列中的PAUSE_TIMEOUT_MSG消息，因为Launcher组件已经在规定的时间内处理完成ActivityManagerService给它发送的中止通知了。

在前面的Step 11中，我们将ActivityStack类的成员变量mPausingActivity指向了与Launcher组件对应的ActivityRecord对象，因此，第18行的if语句就为true。接下来第19行将这个ActivityRecord对象的成员变量state的值设置为ActivityState.PAUSED，表示Launcher组件已经进入Paused状态了。最后第20行调用成员函数completePauseLocked来执行启动MainActivity组件的操作。

Step 20：ActivityStack.completePauseLocked

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02     ......
03
04     private final void completePauseLocked() {
05         ......
06
07         ActivityRecord prev = mPausingActivity;
08         ......
09
10         if (prev != null) {
11             ......
12
13             mPausingActivity = null;
14         }
15
16         if (!mService.mSleeping && !mService.mShuttingDown) {
17             resumeTopActivityLocked(prev);
18         }
19
20         ......
21     }
22
23     ......
24 }
```

第7行首先将ActivityRecord对象prev指向ActivityStack类的成员变量mPausingActivity，接着第13行将后者设置为null，表示系统当前正在中止的Activity组件已经进入Paused状态了。

第16行检查系统当前是不是正在进入睡眠或者关闭状态。如果不是，那么第17行就调用成员函数resumeTopActivityLocked来启动位于Activity组件堆栈顶端的Activity组件。

Step 21：ActivityStack.resumeTopActivityLocked

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02     ......
03
04     final boolean resumeTopActivityLocked(ActivityRecord prev) {
05         // Find the first activity that is not finishing.
06         ActivityRecord next = topRunningActivityLocked(null);
07         .......
08
09         // We need to start pausing the current activity so the top one
10         // can be resumed...
11         if (mResumedActivity != null) {
12             ......
13             startPausingLocked(userLeaving, false);
14             return true;
15         }
16
17         ......
18
19         if (next.app != null && next.app.thread != null) {
```

```
20              ......
21          } else {
22              ......
23              startSpecificActivityLocked(next, true, true);
24          }
25
26          return true;
27      }
28
29      ......
30  }
```

在前面的Step 10中，ActivityManagerService已经调用过它的成员函数resumeTopActivityLocked来试图启动MainActivity组件了；但是由于那时候Launcher组件尚未进入Paused状态，即ActivityStack类的成员变量mResumedActivity不等于null，因此，就会先调用成员函数startPausingLocked来执行中止Launcher组件的操作。

在前面的Step 11中，ActivityManagerService在向Launcher组件发送中止通知之前，已经将ActivityStack类的成员变量mResumedActivity设置为null了，因此，这时候就会跳过第11行到第15行的if语句块。

第6行调用成员函数topRunningActivityLocked获得的是位于Activity组件堆栈顶端的，与即将启动的MainActivity组件对应的一个ActivityRecord对象next。由于这时候MainActivity组件尚未被启动起来，因此，它的成员变量app就会等于null，接下来第23行就会调用成员函数startSpecificActivityLocked将它启动起来。

Step 22：ActivityStack.startSpecificActivityLocked

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02      ......
03
04      private final void startSpecificActivityLocked(ActivityRecord r,
05              boolean andResume, boolean checkConfig) {
06          // Is this activity's application already running?
07          ProcessRecord app = mService.getProcessRecordLocked(r.processName,
08              r.info.applicationInfo.uid);
09
10          ......
11
12          if (app != null && app.thread != null) {
13              try {
14                  realStartActivityLocked(r, app, andResume, checkConfig);
15                  return;
16              } catch (RemoteException e) {
17                  ......
18              }
19          }
20
21          mService.startProcessLocked(r.processName, r.info.applicationInfo, true, 0,
22              "activity", r.intent.getComponent(), false);
23      }
24
25      ......
26  }
```

在ActivityManagerService中，每一个Activity组件都有一个用户ID和一个进程名称；其中，用户ID是在安装该Activity组件时由PackageManagerService分配的，而进程名称则是由该Activity组件的android:process属性来决定的。ActivityManagerService在启动一个Activity组件时，首先会以它的用户ID

和进程名称来检查系统中是否存在一个对应的应用程序进程。如果存在，就会直接通知这个应用程序进程将该Activity组件启动起来；否则，就会先以这个用户ID和进程名称来创建一个应用程序进程，然后再通知这个应用程序进程将该Activity组件启动起来。

第7行和第8行代码在ActivityManagerService中检查与ActivityRecord 对象r对应的Activity组件所需要的应用程序进程是否已经存在。如果存在，第14行就直接调用成员函数realStartActivityLocked来启动该Activity组件；否则，第21行就会先调用ActivityManagerService类的成员函数startProcessLocked为该Activity组件创建一个应用程序进程，然后再将它启动起来。

由于MainActivity组件是第一次被启动，即这时候系统中是不可能存在所需要的应用程序进程的，因此，接下来就会调用ActivityManagerService类的成员函数startProcessLocked来为MainActivity组件创建一个应用程序进程。

Step 23：ActivityManagerService.startProcessLocked

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     final ProcessRecord startProcessLocked(String processName,
06             ApplicationInfo info, boolean knownToBeDead, int intentFlags,
07             String hostingType, ComponentName hostingName, boolean allowWhileBooting) {
08         ProcessRecord app = getProcessRecordLocked(processName, info.uid);
09         ......
10
11         String hostingNameStr = hostingName != null
12             ? hostingName.flattenToShortString() : null;
13         ......
14
15         if (app == null) {
16             app = newProcessRecordLocked(null, info, processName);
17             mProcessNames.put(processName, info.uid, app);
18         }
19         ......
20
21         startProcessLocked(app, hostingType, hostingNameStr);
22         return (app.pid != 0) ? app : null;
23     }
24
25     ......
26 }
```

第8行代码首先检查请求创建的应用程序进程是否已经存在了。如果不存在，第16行就会根据指定的名称以及用户ID来创建一个ProcessRecord对象，并且第17行将它保存在ActivityManagerService类的成员变量mProcessNames中。

最后，第21行就调用另外一个重载版本的成员函数startProcessLocked来创建一个应用程序进程，它的实现如下所示。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02          implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03      ......
04
05      private final void startProcessLocked(ProcessRecord app,
06                  String hostingType, String hostingNameStr) {
07          ......
08
09          try {
```

```
10              int uid = app.info.uid;
11              int[] gids = null;
12              try {
13                  gids = mContext.getPackageManager().getPackageGids(
14                      app.info.packageName);
15              } catch (PackageManager.NameNotFoundException e) {
16                  ......
17              }
18              ......
19
20              int debugFlags = 0;
21              ......
22
23              int pid = Process.start("android.app.ActivityThread",
24                  mSimpleProcessManagement ? app.processName : null, uid, uid,
25                  gids, debugFlags, null);
26
27              if (pid == 0 || pid == MY_PID) {
28                  ......
29              } else if (pid > 0) {
30                  app.pid = pid;
31                  app.removed = false;
32                  synchronized (mPidsSelfLocked) {
33                     this.mPidsSelfLocked.put(pid, app);
34                     Message msg = mHandler.obtainMessage(PROC_START_TIMEOUT_MSG);
35                     msg.obj = app;
36                     mHandler.sendMessageDelayed(msg, PROC_START_TIMEOUT);
37                  }
38              } else {
39                  ......
40              }
41
42              ......
43          } catch (RuntimeException e) {
44              ......
45          }
46      }
47
48      ......
49  }
```

第10行到第17行代码首先得到要创建的应用程序进程的用户ID和用户组ID[1]，接着第23行到第25行代码再调用Process类的静态成员函数start来启动一个新的应用程序进程[2]。

新的应用程序进程创建成功之后，当前进程就会得到一个大于0的进程ID，保存在变量pid中。接下来第33行就以变量pid为关键字将参数app所指向的一个ProcessRecord对象保存在ActivityManagerService类的成员变量mPidsSelfLocked中。最后第34行到第36行代码向ActivityManagerService所运行在的线程的消息队列中发送一个类型为PROC_START_TIMEOUT_MSG的消息，并且指定这个消息在PROC_START_TIMEOUT毫秒之后处理。新的应用程序进程必须在PROC_START_TIMEOUT毫秒之内完成启动工作，并且向ActivityManagerService发送一个启动完成的通知，以便ActivityManagerService可以在它里面启动一个Activity组件；否则，ActivityManagerService就会认为它启动超时了，因此，就不能将相应的Activity组件启动起来。

前面在调用Process类的静态成员函数start来启动一个新的应用程序进程时，指定该进程的入口函数为android.app.ActivityThread类的静态成员函数main；因此，接下来我们就从它开始分析新的应用程

1　应用程序进程的用户ID和用户组ID的分配过程可以参考第16章——Android应用程序的安装和显示过程。

2　应用程序进程的启动过程可以参考第12章——Android应用程序进程的启动过程。

序进程的启动过程，如图7-7所示。

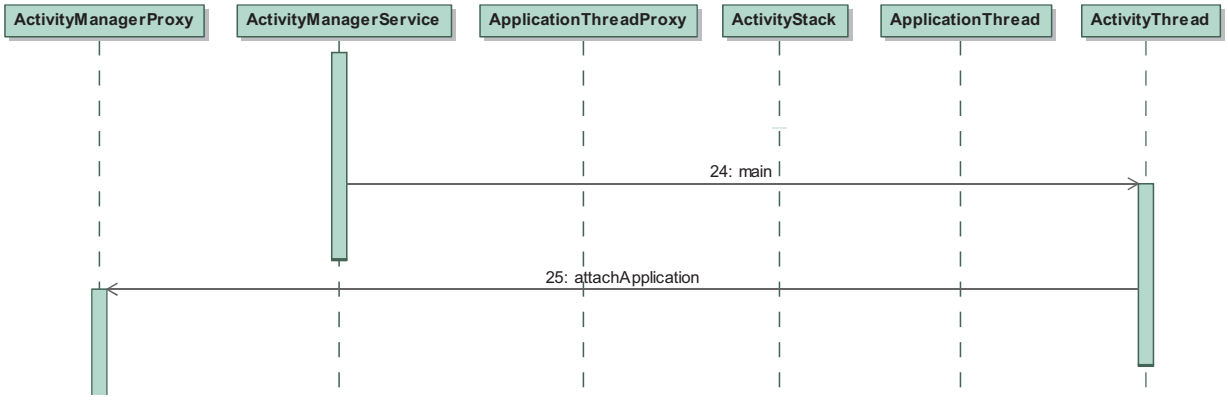


图7-7　MainActivity组件的启动过程——Step 24和Step 25

Step 24：ActivityThread.main

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     final ApplicationThread mAppThread = new ApplicationThread();
05     ......
06
07     private final void attach(boolean system) {
08         ......
09
10         mSystemThread = system;
11         if (!system) {
12             ......
13
14             IActivityManager mgr = ActivityManagerNative.getDefault();
15             try {
16                 mgr.attachApplication(mAppThread);
17             } catch (RemoteException ex) {
18             }
19         }
20         ......
21     }
22
23     ......
24
25     public static final void main(String[] args) {
26         .......
27
28         Looper.prepareMainLooper();
29         ......
30
31         ActivityThread thread = new ActivityThread();
32         thread.attach(false);
33         ......
```

```
34
35          Looper.loop();
36          .......
37      }
38  }
```

新的应用程序进程在启动时，主要做了两件事情。

第一件事情是在进程中创建一个ActivityThread对象，并且调用它的成员函数attach向ActivityManagerService发送一个启动完成通知。

第二件事情是调用Looper类的静态成员函数prepareMainLooper创建一个消息循环，并且在向ActivityManagerService发送启动完成通知之后，使得当前进程进入到这个消息循环中。

我们主要关注新的应用程序是如何向ActivityManagerService发送一个启动完成通知的。

第31行在创建ActivityThread对象thread时，会同时在它内部创建一个ApplicationThread对象mAppThread。前面提到，ActivityThread对象内部的ApplicationThread对象是一个Binder本地对象，ActivityManagerService就是通过它来和应用程序进程通信的。

在ActivityThread类的成员函数attach中，第14行首先调用ActivityManagerNative类的静态成员函数getDefault来获得ActivityManagerService的一个代理对象；然后第16行调用它的成员函数attachApplication向ActivityManagerService发送一个进程间通信请求，并且将前面所创建的ApplicationThread对象传递给ActivityManagerService。

ActivityManagerService代理对象的类型为ActivityManagerProxy，因此，接下来就会调用ActivityManagerProxy类的成员函数向ActivityManagerService发送一个进程间通信请求。

Step 25：ActivityManagerProxy.attachApplication

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
01 class ActivityManagerProxy implements IActivityManager
02 {
03     ......
04
05     public void attachApplication(IApplicationThread app) throws RemoteException
06     {
07         Parcel data = Parcel.obtain();
08         Parcel reply = Parcel.obtain();
09         data.writeInterfaceToken(IActivityManager.descriptor);
10         data.writeStrongBinder(app.asBinder());
11         mRemote.transact(ATTACH_APPLICATION_TRANSACTION, data, reply, 0);
12         reply.readException();
13         data.recycle();
14         reply.recycle();
15     }
16
17     ......
18 }
```

第9行和第10行代码将从前面传进来的参数写入到Parcel对象data中，然后再通过ActivityManagerProxy类内部的一个Binder代理对象mRemote向ActivityManagerService发送一个类型为ATTACH_APPLICATION_TRANSACTION的进程间通信请求。

以上2步都是在新的应用程序进程中执行的，接下来第26步到第29步是在ActivityManagerService中执行的，主要是用来处理新的应用程序进程发出的类型为ATTACH_APPLICATION_TRANSACTION的进程间通信请求，如图7-8所示。

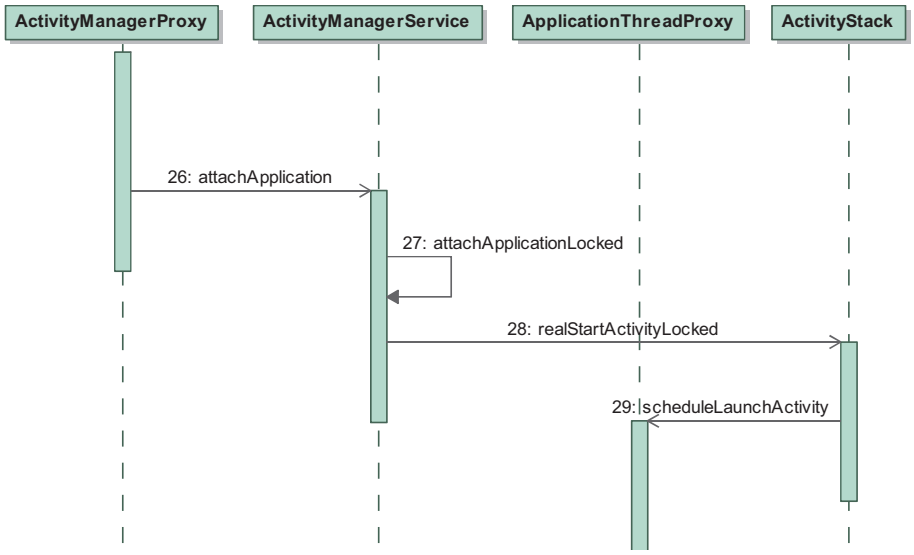


图7-8　MainActivity组件的启动过程——Step 26至Step 29

Step 26：ActivityManagerService.attachApplication

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     public final void attachApplication(IApplicationThread thread) {
06         synchronized (this) {
07             int callingPid = Binder.getCallingPid();
08             final long origId = Binder.clearCallingIdentity();
09             attachApplicationLocked(thread, callingPid);
10             Binder.restoreCallingIdentity(origId);
11         }
12     }
13
14     ......
15 }
```

ActivityManagerService类的成员函数attachApplication接收到新的应用程序进程发送过来的类型为ATTACH_APPLICATION_TRANSACTION的进程间通信请求之后，它就知道新的应用程序进程已经启动完成了。因此，接下来第9行就调用成员函数attachApplicationLocked来继续执行启动MainActivity组件的操作。

Step 27：ActivityManagerService.attachApplicationLocked

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     private final boolean attachApplicationLocked(IApplicationThread thread,
06             int pid) {
07         // Find the application record that is being attached...  either via
08         // the pid if we are running in multiple processes, or just pull the
09         // next app record if we are emulating process with anonymous threads.
10         ProcessRecord app;
11         if (pid != MY_PID && pid >= 0) {
12             synchronized (mPidsSelfLocked) {
```

```
13                  app = mPidsSelfLocked.get(pid);
14              }
15          }
16          ......
17
18          String processName = app.processName;
19          ......
20
21          app.thread = thread;
22          app.curAdj = app.setAdj = -100;
23          app.curSchedGroup = Process.THREAD_GROUP_DEFAULT;
24          app.setSchedGroup = Process.THREAD_GROUP_BG_NONINTERACTIVE;
25          app.forcingToForeground = null;
26          app.foregroundServices = false;
27          app.debugging = false;
28
29          mHandler.removeMessages(PROC_START_TIMEOUT_MSG, app);
30
31          boolean normalMode = mProcessesReady || isAllowedWhileBooting(app.info);
32          ......
33
34          // See if the top visible activity is waiting to run in this process...
35          ActivityRecord hr = mMainStack.topRunningActivityLocked(null);
36          if (hr != null && normalMode) {
37              if (hr.app == null && app.info.uid == hr.info.applicationInfo.uid
38                  && processName.equals(hr.processName)) {
39                  try {
40                      if (mMainStack.realStartActivityLocked(hr, app, true, true)) {
41                          ......
42                      }
43                  } catch (Exception e) {
44                      ......
45                  }
46              }
47              ......
48          }
49          ......
50
51          return true;
52      }
53
54      ......
55  }
```

参数pid指向了前面所创建的应用程序进程的PID。在前面的Step 23中，ActivityManagerService以这个PID为关键字将一个ProcessRecord对象保存在了成员变量mPidsSelfLocked中。因此，第10行到第15行代码就可以通过参数pid将这个ProcessRecord对象取回来，并且保存在变量app中。

前面得到的ProcessRecord对象app就是用来描述新创建的应用程序进程的。现在既然这个应用程序进程已经启动起来了，第21行到第27行代码就继续对ProcessRecord对象app进行初始化，其中，最重要的是将它的成员变量thread设置为参数thread所指向的一个ApplicationThread代理对象。这样，ActivityManagerService以后就可以通过这个ApplicationThread代理对象来和新创建的应用程序进程进行通信了。

第29行删除ActivityManagerService所运行在的线程的消息队列中的PROC_START_TIMEOUT_MSG消息，因为新的应用程序进程已经在规定的时间内启动起来了。

第35行得到位于Activity组件堆栈顶端的一个ActivityRecord对象hr，与它对应的Activity组件就是即将要启动的MainActivity组件，接着第37行的if语句检查这个Activity组件的用户ID和进程名称是否与ProcessRecord对象app所描述的应用程序进程的用户ID和进程名称一致。如果一致，那么就说明

ActivityRecord对象hr所描述的Activity组件是应该在ProcessRecord对象app所描述的应用程序进程中启动的，因此，第40行就会调用ActivityStack类的成员函数realStartActivityLocked来请求该应用程序进程启动一个Activity组件。

Step 28：ActivityStack.realStartActivityLocked

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02     ......
03
04     final boolean realStartActivityLocked(ActivityRecord r,
05             ProcessRecord app, boolean andResume, boolean checkConfig)
06             throws RemoteException {
07         ......
08
09         r.app = app;
10         ......
11
12         int idx = app.activities.indexOf(r);
13         if (idx < 0) {
14             app.activities.add(r);
15         }
16         ......
17
18         try {
19             ......
20
21             List<ResultInfo> results = null;
22             List<Intent> newIntents = null;
23             if (andResume) {
24                 results = r.results;
25                 newIntents = r.newIntents;
26             }
27             ......
28
29             app.thread.scheduleLaunchActivity(new Intent(r.intent), r,
30                 System.identityHashCode(r),
31                 r.info, r.icicle, results, newIntents, !andResume,
32                 mService.isNextTransitionForward());
33             ......
34         } catch (RemoteException e) {
35             ......
36         }
37         ......
38
39         return true;
40     }
41
42     ......
43 }
```

第9行首先将参数r的成员变量app的值设置为参数app，表示它描述的Activity组件是在参数app所描述的应用程序进程中启动的，接着第14行又将该Activity组件添加到参数app所描述的应用程序进程的Activity组件列表中。

完成前面的一些准备工作之后，最后第29行到第32行就调用参数app的成员变量thread的成员函数scheduleLaunchActivity，来通知前面创建的应用程序进程启动由参数r所描述的一个Activity组件，即MainActivity组件。

参数app的成员变量thread是一个类型为ApplicationThreadProxy的Binder代理对象，因此，接下来就会调用ApplicationThreadProxy类的成员函数scheduleLaunchActivity来向前面创建的应用程序进程发送一个进程间通信请求。

Step 29：ApplicationThreadProxy.scheduleLaunchActivity

frameworks/base/core/java/android/app/ApplicationThreadNative.java

```
01 class ApplicationThreadProxy implements IApplicationThread {
02     ......
03
04     public final void scheduleLaunchActivity(Intent intent, IBinder token, int ident,
05             ActivityInfo info, Bundle state, List<ResultInfo> pendingResults,
06             List<Intent> pendingNewIntents, boolean notResumed, boolean isForward)
07             throws RemoteException {
08         Parcel data = Parcel.obtain();
09         data.writeInterfaceToken(IApplicationThread.descriptor);
10         intent.writeToParcel(data, 0);
11         data.writeStrongBinder(token);
12         data.writeInt(ident);
13         info.writeToParcel(data, 0);
14         data.writeBundle(state);
15         data.writeTypedList(pendingResults);
16         data.writeTypedList(pendingNewIntents);
17         data.writeInt(notResumed ? 1 : 0);
18         data.writeInt(isForward ? 1 : 0);
19         mRemote.transact(SCHEDULE_LAUNCH_ACTIVITY_TRANSACTION, data, null,
20             IBinder.FLAG_ONEWAY);
21         data.recycle();
22     }
23
24     ......
25 }
```

第9行到第18行代码将从前面传进来的参数写入到Parcel对象data中，接着再通过ApplicationThreadProxy类内部的一个Binder代理对象mRemote向前面创建的应用程序进程发送一个类型为SCHEDULE_LAUNCH_ACTIVITY_TRANSACTION的进程间通信请求。

以上4步都是在ActivityManagerService中执行的，接下来第30步到第35步是在前面创建的应用程序进程中执行的，主要用来处理ActivityManagerService发出的类型为SCHEDULE_LAUNCH_ACTIVITY_TRANSACTION的进程间通信请求，如图7-9所示。

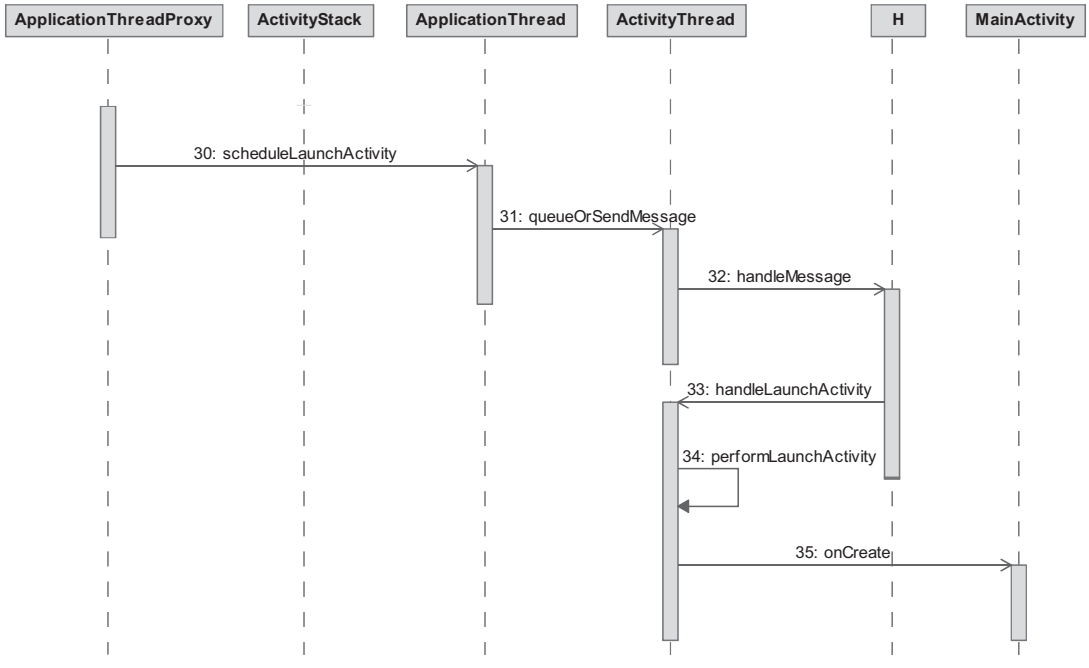


图7-9　MainActivity组件的启动过程——Step 30至Step 35

Step 30：ApplicationThread.scheduleLaunchActivity

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final class ApplicationThread extends ApplicationThreadNative {
05         ......
06
07         // we use token to identify this activity without having to send the
08         // activity itself back to the activity manager. (matters more with ipc)
09         public final void scheduleLaunchActivity(Intent intent, IBinder token, int ident,
10                 ActivityInfo info, Bundle state, List<ResultInfo> pendingResults,
11                 List<Intent> pendingNewIntents, boolean notResumed, boolean isForward) {
12             ActivityClientRecord r = new ActivityClientRecord();
13
14             r.token = token;
15             r.ident = ident;
16             r.intent = intent;
17             r.activityInfo = info;
18             r.state = state;
19
20             r.pendingResults = pendingResults;
21             r.pendingIntents = pendingNewIntents;
22
23             r.startsNotResumed = notResumed;
24             r.isForward = isForward;
25
26             queueOrSendMessage(H.LAUNCH_ACTIVITY, r);
27         }
28
29         ......
30     }
31
32     ......
33 }
```

ApplicationThread类的成员函数scheduleLaunchActivity用来处理类型为SCHEDULE_LAUNCH_ACTIVITY_TRANSACTION的进程间通信请求，它主要是将要启动的Activity组件的信息封装成一个ActivityClientRecord对象，然后再以这个ActivityClientRecord对象为参数来调用ActivityThread类的成员函数queueOrSendMessage，以便可以往新创建的应用程序进程的主线程的消息队列发送一个类型为LAUNCH_ACTIVITY的消息。

Step 31：ActivityThread.queueOrSendMessage

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     // if the thread hasn't started yet, we don't have the handler, so just
05     // save the messages until we're ready.
06     private final void queueOrSendMessage(int what, Object obj) {
07         queueOrSendMessage(what, obj, 0, 0);
08     }
09
10     ......
11
12     private final void queueOrSendMessage(int what, Object obj, int arg1, int arg2) {
13         synchronized (this) {
14             ......
15             Message msg = Message.obtain();
```

```
16              msg.what = what;
17              msg.obj = obj;
18              msg.arg1 = arg1;
19              msg.arg2 = arg2;
20              mH.sendMessage(msg);
21          }
22      }
23
24      ......
25  }
```

第15行到第19行代码将前面Step 30传进来的一个ActivityClientRecord对象封装成一个Message对象msg，接着第20行调用ActivityThread类的成员变量mH的成员函数sendMessage向应用程序进程的主线程的消息队列发送一个类型为LAUNCH_ACTIVITY的消息，以便它可以执行启动MainActivity组件的操作。这个消息最终是在ActivityThread类的成员变量mH的成员函数handleMessage中处理的。

Step 32：H.handleMessage

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02      ......
03
04      private final class H extends Handler {
05          ......
06
07          public void handleMessage(Message msg) {
08              ......
09              switch (msg.what) {
10              case LAUNCH_ACTIVITY: {
11                  ActivityClientRecord r = (ActivityClientRecord)msg.obj;
12
13                  r.packageInfo = getPackageInfoNoCheck(
14                      r.activityInfo.applicationInfo);
15                  handleLaunchActivity(r, null);
16              } break;
17              ......
18              }
19
20              ......
21          }
22
23          ......
24      }
25
26      ......
27  }
```

第11行代码首先将Message对象msg的成员变量obj转换成一个ActivityClientRecord对象r，接着第13行和第14行调用ActivityThread类的成员函数getPackageInfoNoCheck来获得一个LoadedApk对象，并且保存在ActivityClientRecord对象r的成员变量packageInfo中。

我们知道，每一个Android应用程序都是打包在一个Apk文件中的。一个Apk文件包含了一个Android应用程序的所有资源，应用程序进程在启动一个Activity组件时，需要将它所属的Apk文件加载进来，以便可以访问它里面的资源。在ActivityThread类内部，就使用一个LoadedApk对象来描述一个已加载的Apk文件。

最后，第15行调用ActivityThread类的成员函数handleLaunchActivity来启动由ActivityClientRecord对象r所描述的一个Activity组件，即MainActivity组件。

Step 33：ActivityThread.handleLaunchActivity

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final void handleLaunchActivity(ActivityClientRecord r, Intent customIntent) {
05         ......
06
07         Activity a = performLaunchActivity(r, customIntent);
08
09         if (a != null) {
10             r.createdConfig = new Configuration(mConfiguration);
11             Bundle oldState = r.state;
12             handleResumeActivity(r.token, false, r.isForward);
13
14             ......
15         }
16         ......
17     }
18
19     ......
20 }
```

第7行首先调用成员函数performLaunchActivity将MainActivity组件启动起来，接着第12行调用成员函数handleResumeActivity将MainActivity组件的状态设置为Resumed，表示它是系统当前激活的Activity组件。

接下来，我们就主要关注MainActivity组件的启动过程，即ActivityThread类的成员函数performLaunchActivity的实现。

Step 34：ActivityThread.performLaunchActivity

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent)
{
05         ......
06
07         ComponentName component = r.intent.getComponent();
08         ......
09
10         Activity activity = null;
11         try {
12             java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
13             activity = mInstrumentation.newActivity(
14                 cl, component.getClassName(), r.intent);
15             ......
16         } catch (Exception e) {
17             ......
18         }
19
20         try {
21             Application app = r.packageInfo.makeApplication(false, mInstrumentation);
22
23             ......
24
25             if (activity != null) {
26                 ContextImpl appContext = new ContextImpl();
27                 appContext.init(r.packageInfo, r.token, this);
```

```
28                  appContext.setOuterContext(activity);
29                  CharSequence title = r.activityInfo.loadLabel(appContext.getPackageManager());
30                  Configuration config = new Configuration(mConfiguration);
31                  ......
32                  activity.attach(appContext, this, getInstrumentation(), r.token,
33                      r.ident, app, r.intent, r.activityInfo, title, r.parent,
34                      r.embeddedID, r.lastNonConfigurationInstance,
35                      r.lastNonConfigurationChildInstances, config);
36                  ......
37
38                  mInstrumentation.callActivityOnCreate(activity, r.state);
39
40                  ......
41              }
42              ......
43
44              mActivities.put(r.token, r);
45
46          } catch (SuperNotCalledException e) {
47              ......
48
49          } catch (Exception e) {
50              ......
51          }
52
53          return activity;
54      }
55
56      ......
57  }
```

第7行代码首先获得要启动的Activity组件的包名以及类名，它们使用一个ComponentName对象component来描述。接着第12行到第14行代码将它的类文件加载到内存中，并且创建一个实例。由于所有的Activity组件都是从Activity类继承下来的，因此，我们就可以将前面创建的Activity组件保存在Activity对象activity中。

第26行到第28行代码创建和初始化了一个ContextImpl对象appContext，用来作为前面所创建的Activity对象activity的运行上下文环境，通过它就可以访问到特定的应用程序资源，以及启动其他的应用程序组件。接着第32行到第35行代码使用ContextImpl对象appContext和ActivityClientRecord对象r来初始化Activity对象activity。

Activity对象activity初始化完成之后，第38行就调用成员变量mInstrumentation的成员函数callActivityOnCreate将Activity对象activity启动起来。在这个过程中，Activity对象activity的成员函数onCreate就会被调用。一般来说，我们在自定义一个Activity组件时，都会重写其父类Activity的成员函数onCreate，以便可以加载自己的用户界面，以及执行其他一些业务相关的初始化工作。

Activity对象activity启动完成之后，第44行就会以ActivityClientRecord对象r的成员变量token为关键字，将ActivityClientRecord对象r保存在ActivityThread类的成员变量mActivities中。

注意

ActivityClientRecord对象r的成员变量token是一个Binder代理对象，它指向了ActivityManagerService内部的一个ActivityRecord对象。这个ActivityRecord对象和ActivityClientRecord对象r一样，都是用来描述前面启动的Activity组件的，只不过前者是在ActivityManagerService中使用，而后者是在应用程序进程中使用。

由于Activity对象activity指向的是一个MainActivity组件，因此，启动MainActivity组件的最后一步便是调用它的成员函数onCreate，以便让它有机会执行自定义的初始化工作。

Step 35：MainActivity.onCreate

packages/experimental/Activity/src/shy/luo/activity/MainActivity.java

```
01 public class MainActivity extends Activity  implements OnClickListener {
02     ......
03
04     @Override
05     public void onCreate(Bundle savedInstanceState) {
06         super.onCreate(savedInstanceState);
07         setContentView(R.layout.main);
08
09         startInProcessButton = (Button)findViewById(R.id.button_start_in_process);
10         startInNewProcessButton = (Button)findViewById(R.id.button_start_in_new_process);
11
12         startInProcessButton.setOnClickListener(this);
13         startInNewProcessButton.setOnClickListener(this);
14
15         Log.i(LOG_TAG, "Main Activity Created.");
16     }
17
18     ......
19 }
```

MainActivity组件的成员函数onCreate主要用来加载用户界面，以及对用户界面上的控件进行初始化。

至此，MainActivity组件的启动过程就分析完了。MainActivity组件作为应用程序Activity的根Activity，它启动起来之后，就意味着应用程序Activity启动起来了。因此，我们可以将一个根Activity的启动过程看作一个Android应用程序的启动过程。

7.3 子Activity组件在进程内的启动过程

本节将以应用程序Activity的SubActivityInProcess组件的启动过程为例，来说明Android应用程序的子Activity组件在进程内的启动过程。

在进程内启动的子Activity组件与启动它的Activity组件是运行在同一个应用程序进程中的。在前面的7.1节中，当SubActivityInProcess组件被MainActivity组件启动起来之后，我们就可以通过adb shell dumpsys activity命令来查看它的信息，如下所示。

```
USER@MACHINE:~/Android$ adb shell dumpsys activity
  ......
  Running activities (most recent first):
    TaskRecord{4072bf20 #3 A shy.luo.activity}
      Run #2: HistoryRecord{406935b0 shy.luo.activity/.SubActivityInProcess}
      Run #1: HistoryRecord{40730138 shy.luo.activity/.MainActivity}
    TaskRecord{405e6360 #2 A com.android.launcher}
      Run #0: HistoryRecord{40558328 com.android.launcher/com.android.launcher2.Launcher}
......
  Running processes (most recent first):
    ......
    Proc # 2: adj=fore /F 4070b7d8 383:shy.luo.activity.mainprocess/10036 (top-activity)
    ......
  ......
```

从Running activities小节的信息可以看出，系统的Activity组件堆栈有TaskRecord#2和TaksRecord#3两个任务；其中，任务TaskRecord#2包含了一个Launcher组件，任务TaksRecord#3包含了一个MainActivity组件和一个SubActivityInProcess组件。在任务TaksRecord#3中，由于SubActivityInProcess组

件是由MainActivity组件启动起来的，因此，前者就位于后者的上面。

由于ManActivity组件和SubActivityInProcess组件的android:process属性都被设置为“shy.luo.activity.mainporcess”，因此，ActivityManagerService在启动SubActivityInProcess组件时，就会发现系统中已经存在一个名称为“shy.luo.activity.mainporcess”的应用程序进程了，接下来它就直接在这个应用程序进程中将SubActivityInProcess组件启动起来。我们可以从Running processes小节的信息来证实这一点。

MainActivity组件启动SubActivityInProcess组件的过程如下所示。

（1）MainActivity组件向ActivityManagerService发送一个启动SubActivityInProcess组件的进程间通信请求。

（2）ActivityManagerService首先将要启动的SubActivityInProcess组件的信息保存下来，然后再向MainActivity组件发送一个进入中止状态的进程间通信请求。

（3）MainActivity组件进入到中止状态之后，就会向ActivityManagerService发送一个已进入中止状态的进程间通信请求，以便ActivityManagerService可以继续执行启动SubActivityInProcess组件的操作。

（4）ActivityManagerService发现用来运行SubActivityInProcess组件的应用程序进程已经存在，因此它就会将第2步保存下来的SubActivityInProcess组件信息发送给该应用程序进程，以便它可以将SubActivityInProcess组件启动起来。

由于ActivityManagerService不需要创建一个新的应用程序进程来启动SubActivityInProcess组件，因此，SubActivityInProcess组件的启动过程要比MainActivity组件的启动过程简单一些，一共分为30个步骤。接下来，我们就详细分析每一个步骤。

第1步到第5步如图7-10所示。

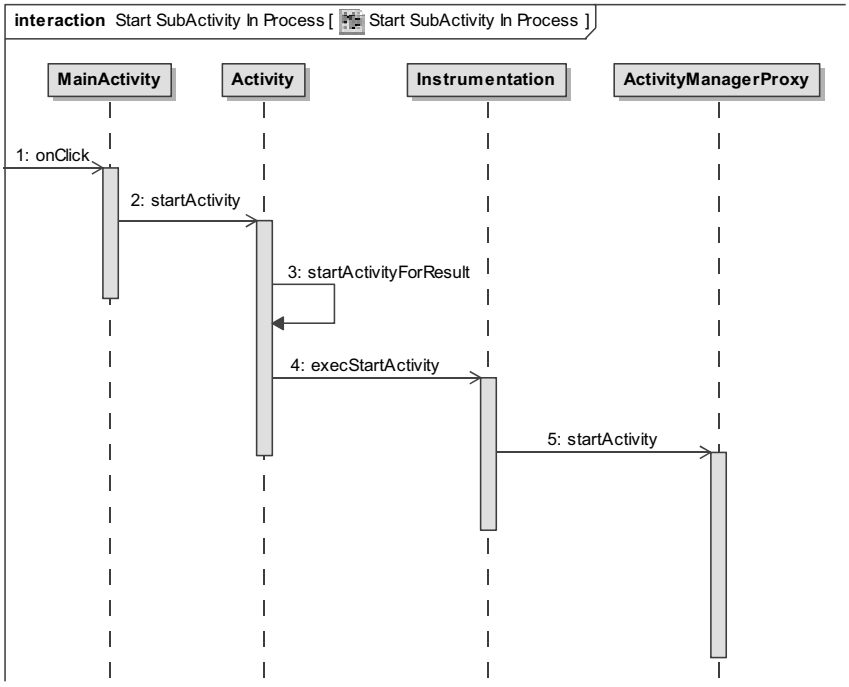


图7-10　SubActivityInProcess组件的启动过程——Step 1至Step 5

Step 1：MainActivity.onClick

packages/experimental/Activity/src/shy/luo/activity/MainActivity.java

```
01 public class MainActivity extends Activity  implements OnClickListener {
02     ......
03
04     @Override
```

```
05     public void onClick(View v) {
06         if(v.equals(startInProcessButton)) {
07                 Intent intent = new Intent("shy.luo.activity.subactivity.in.process");
08                 startActivity(intent);
09         } else if(v.equals(startInNewProcessButton)) {
10                 ......
11         }
12     }
13 }
```

第7行的代码首先创建了一个Action名称为“shy.luo.activity.subactivity.in.process”的Intent对象，接着第8行再以这个Intent对象为参数来调用父类Activity的成员函数startActivity，以便可以隐式地启动一个名称为“shy.luo.activity.subactivity.in.process”的Activity组件。

由于应用程序Activity的SubActivityInProcess组件的Action名称被设置为“shy.luo.activity.subactivity.in.process”，因此，接下来实际上要启动的Activity组件便是SubActivityInProcess组件了。

Step 2：Activity.startActivity

Step 3：Activity.startActivityForResult

Step 4：Instrumentation.execStartActivity

Step 5：ActivityManagerProxy.startActivity

第2步到5步可以参考7.2节中的第2步到第5步。

以上5步都是在应用程序Activity中执行的，接下来第6步到第12步是在ActivityManagerService中执行的，主要用来处理MainActivity组件发出的类型为START_ACTIVITY_TRANSACTION的进程间通信请求，即通知ActivityManagerService将SubActivityInProcess组件启动起来，如图7-11所示。

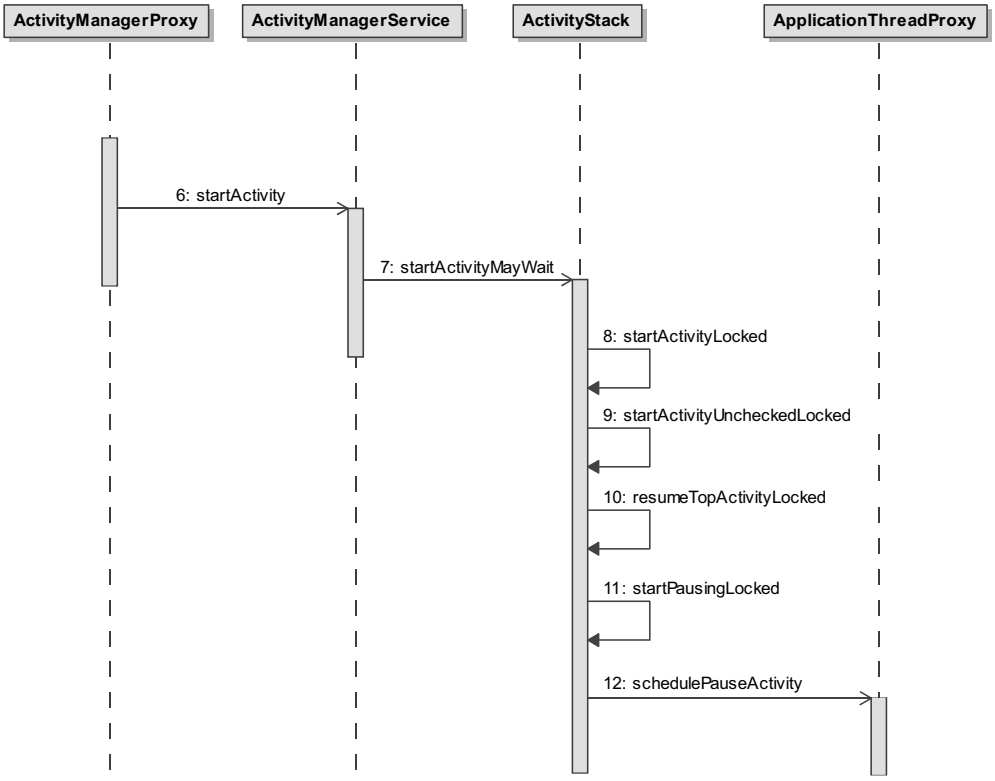


图7-11　SubActivityInProcess组件的启动过程——Step 6至Step 12

Step 6：ActivityManagerService.startActivity

Step 7：ActivityStack.startActivityMayWait

Step 8：ActivityStack.startActivityLocked

第6步到第8步可以参考7.2节中的第6步到第8步。

Step 9：ActivityStack.startActivityUncheckedLocked

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02     ......
03
04     final int startActivityUncheckedLocked(ActivityRecord r,
05             ActivityRecord sourceRecord, Uri[] grantedUriPermissions,
06             int grantedMode, boolean onlyIfNeeded, boolean doResume) {
07         ......
08
09         int launchFlags = intent.getFlags();
10         ......
11
12         boolean addingToTask = false;
13         ......
14
15         boolean newTask = false;
16
17         // Should this be considered a new task?
18         if (r.resultTo == null && !addingToTask
19             && (launchFlags&Intent.FLAG_ACTIVITY_NEW_TASK) != 0) {
20             ......
21
22         } else if (sourceRecord != null) {
23             ......
24             // An existing activity is starting this new activity, so we want
25             // to keep the new one in the same task as the one that is starting
26             // it.
27             r.task = sourceRecord.task;
28             ......
29
30         } else {
31             ......
32         }
33
34         ......
35
36         startActivityLocked(r, newTask, doResume);
37         return START_SUCCESS;
38     }
39
40     ......
41 }
```

从前面7.2节中的Step 9可以知道，ActivityRecord对象r用来描述即将要启动的SubActivityInProcess组件，而ActivityRecord对象sourceRecord用来描述请求ActivityManagerService启动SubActivityInProcess组件的MainActivity组件。

在前面的Step 1中，由于MainActivity组件没有指定要在新的任务中启动SubActivityInProcess组件，并且MainActivity组件和SubActivityInProcess组件的android:taskAffinity属性都是一样的，即ActivityManagerService不需要创建一个新的任务来启动SubActivityInProcess组件，因此，第18行和第22行的if语句就分别为false和true。接下来第27行就会将ActivityRecord对象sourceRecord的成员变量task的

值赋给ActivityRecord对象r的成员变量task，表示SubActivityInProcess组件和MainActivity组件是运行在同一个任务中的。

最后，第36行调用成员函数startActivityLocked来执行启动SubActivityInProcess组件的操作，它的实现如下所示。

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02     ......
03
04     private final void startActivityLocked(ActivityRecord r, boolean newTask,
05             boolean doResume) {
06         final int NH = mHistory.size();
07
08         int addPos = -1;
09
10         if (!newTask) {
11             // If starting in an existing task, find where that is...
12             ......
13             for (int i = NH-1; i >= 0; i--) {
14                 ActivityRecord p = (ActivityRecord)mHistory.get(i);
15                 if (p.finishing) {
16                     continue;
17                 }
18                 if (p.task == r.task) {
19                     // Here it is!  Now, if this is not yet visible to the
20                     // user, then just add it without starting; it will
21                     // get started when the user navigates back to it.
22                     addPos = i+1;
23                     ......
24                     break;
25                 }
26                 ......
27             }
28         }
29         ......
30
31         // Slot the activity into the history stack and proceed
32         mHistory.add(addPos, r);
33         ......
34
35         if (doResume) {
36             resumeTopActivityLocked(null);
37         }
38     }
39
40     ......
41 }
```

由于前面传进来的参数newTask的值等于false，因此，第13行到第27行的for循环首先在Activity组件堆栈mHistory中找到一个合适的位置addPos，接着第32行就可以将参数r所描述的SubActivityInProcess组件保存在这个位置中。

我们知道，在ActivityManagerService中，已经启动的Activity组件是以任务为单位组织在Activity组件堆栈mHistory中的，即同一个任务中的所有Activity组件都是连在一起的，如图7-12所示。

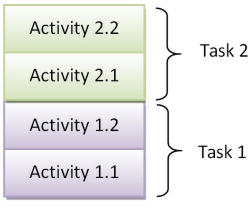


图7-12　Activity组件与任务（Task）的关系

在图7-12中，Activity 1.1和Activity 1.2是属于任务Task 1的，而Activity 2.1和Activity 2.2是属于任务Task 2的，其中，Activity 1.2和Activity 2.2分别是被Activity 1.1和Activity 2.1启动起来的；因此，Activity 1.2位于Activity 1.1的上面，而Activity 2.2位于Activity 2.1的上面。

了解了这个背景知识之后，我们就可以理解第13行到第27行的for循环的逻辑了——首先自上而下地搜索Activity组件堆栈，找到用来运行SubActivityInProcess组件的任务；接着再将这个任务最上面的一个Activity组件的位置i加1，就可以得到SubActivityInProcess组件在Activity组件堆栈mHistory中的位置addPos。

由于前面传进来的参数doResume的值等于true，因此，第36行就会调用成员函数resumeTopActivityLocked来执行启动SubActivityInProcess组件的操作。

Step 10：ActivityStack.resumeTopActivityLocked

Step 11：ActivityStack.startPausingLocked

Step 12：ApplicationThreadProxy.schedulePauseActivity

第10步到第12步可以参考7.2节中的第10步到第12步。

以上7步都是在ActivityManagerService中执行的，接下来第13步到第17步是在应用程序Activity中执行的，主要用来处理ActivityManagerService发出的类型为SCHEDULE_PAUSE_ACTIVITY_TRANSACTION的进程间通信请求，即通知应用程序Activity，它的MainActivity组件要被中止了，如图7-13所示。

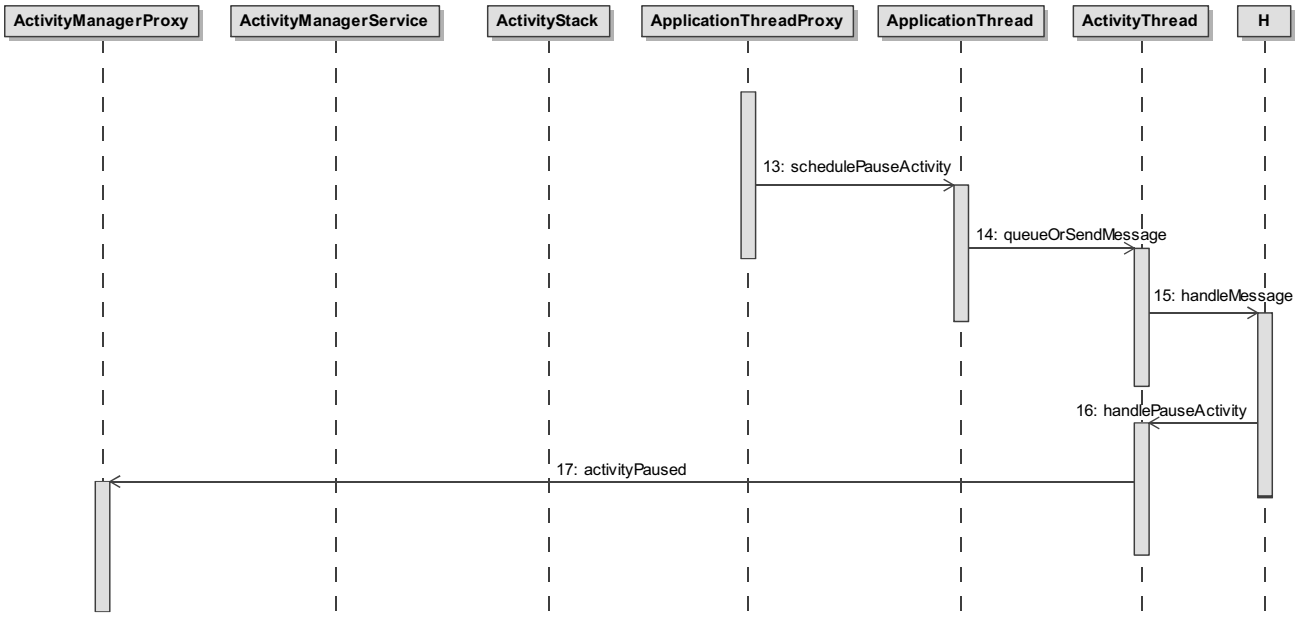


图7-13　SubActivityInProcess组件的启动过程——Step 13至Step 17

Step 13：ApplicationThread.schedulePauseActivity

Step 14：ActivityThread.queueOrSendMessage

Step 15：H.handleMessage

Step 16：ActivityThread.handlePauseActivity

Step 17：ActivityManagerProxy.activityPaused

第13步到第17步可以参考7.2节中的第13步到第17步。

以上5步都是在应用程序Activity中执行的，接下来第18步到第24步是在ActivityManagerService中执行的，主要用来处理MainActivity组件发出的类型为ACTIVITY_PAUSED_TRANSACTION的进程间通信请求，即通知ActivityManagerService，MainActivity组件已经进入到Paused状态，它可以继续执行启动SubActivityInProcess组件的操作了，如图7-14所示。

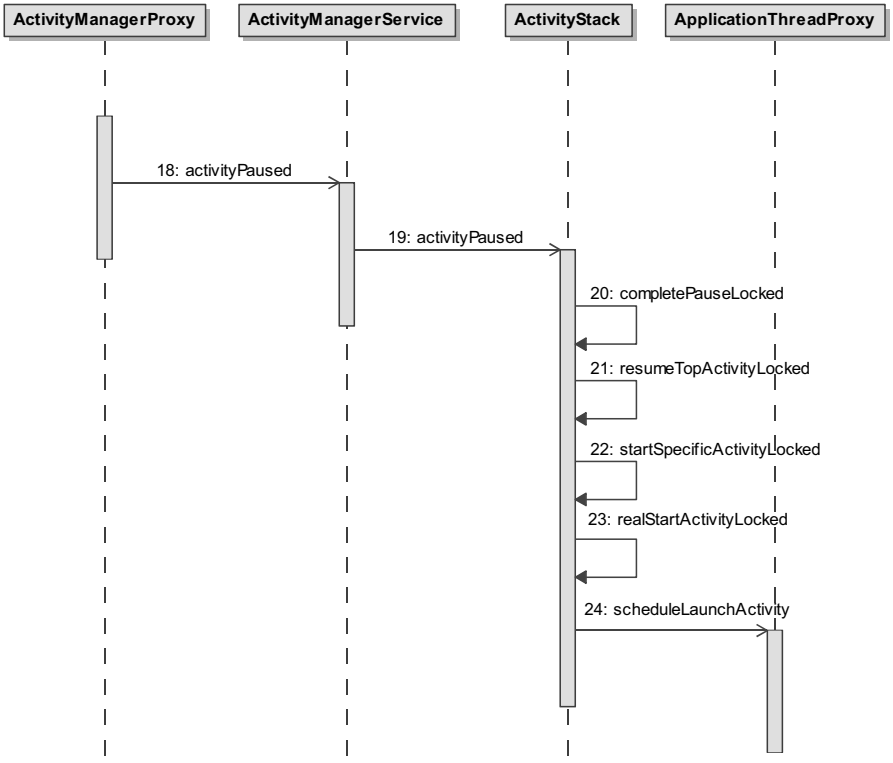


图7-14　SubActivityInProcess组件的启动过程——Step 18至Step 24

Step 18：ActivityManagerService.activityPaused

Step 19：ActivityStack.activityPaused

Step 20：ActivityStack.completePauseLocked

Step 21：ActivityStack.resumeTopActivityLokced

第18步到第21步可以参考7.2节中的第18步到第21步。

Step 22：ActivityStack.startSpecificActivityLocked

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02     ......
03
04     private final void startSpecificActivityLocked(ActivityRecord r,
05             boolean andResume, boolean checkConfig) {
06         // Is this activity's application already running?
07         ProcessRecord app = mService.getProcessRecordLocked(r.processName,
08                 r.info.applicationInfo.uid);
09         ......
10
11         if (app != null && app.thread != null) {
12             try {
13                 realStartActivityLocked(r, app, andResume, checkConfig);
```

```
14                  return;
15              } catch (RemoteException e) {
16                  ......
17              }
18          }
19
20          ......
21      }
22
23      ......
24  }
```

参数r所描述的Activity组件即为SubActivityInProcess组件。由于SubActivityInProcess组件与启动它的MainActivity组件都是属于应用程序Activity的，即它们具有相同的用户ID，并且它们的android:process属性均设置为“shy.luo.activity.mainprocess”，因此，它们就运行在同一个应用程序进程中。

由于MainActivity组件已经启动起来了，即启动SubActivityInProcess组件所需要的应用程序进程已经创建出来了，因此，第7行得到的ProcessRecord对象app就不等于null。ProcessRecord对象app的成员变量thread是一个Binder代理对象，它指向了用来运行MainActivity组件的应用程序进程的一个ApplicationThread对象，因此，第11行的if语句就为true。接下来第13行就会调用成员函数realStartActivityLocked执行启动SubActivityInProcess组件的操作。

Step 23：ActivityStack.realStartActivityLocked

Step 24：ApplicationThreadProxy.scheduleLaunchActivity

第23步和第24步可以参考7.2节中的第28步和第29步。

以上7步都是在ActivityManagerService中执行的，接下来第25步到第30步是在应用程序Activity中执行的，主要用来处理ActivityManagerService发出的类型为SCHEDULE_LAUNCH_ACTIVITY_TRANSACTION的进程间通信请求，即通知MainActivity组件所运行的应用程序进程将SubActivityInProcess组件启动起来，如图7-15所示。

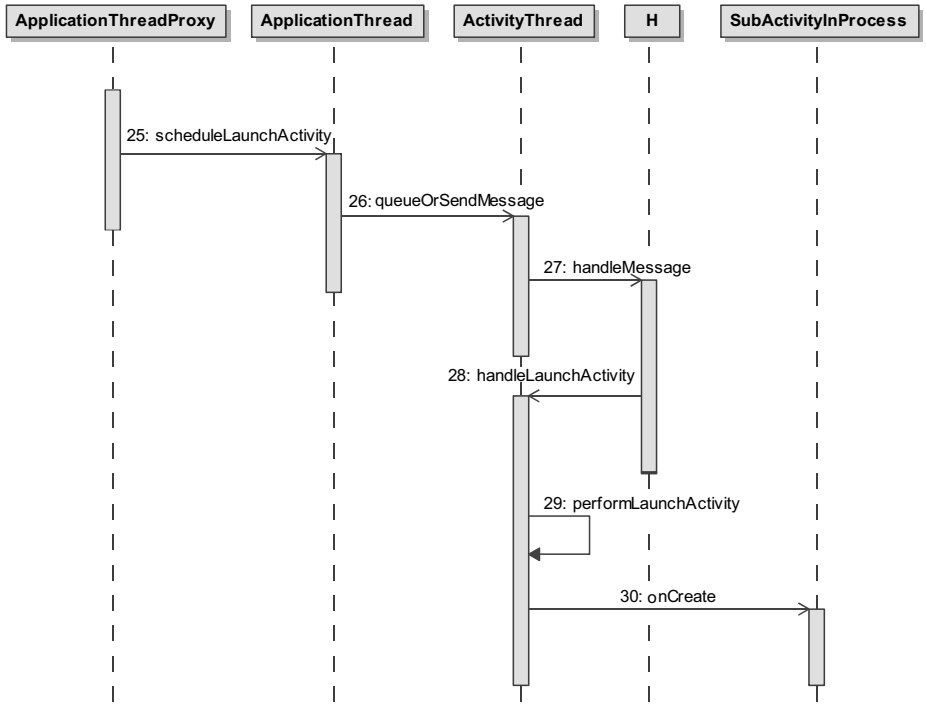


图7-15　SubActivityInProcess组件的启动过程——Step 25至Step 30

Step 25：ApplicationThread.scheduleLaunchActivity

Step 26：ActivityThread.queueOrSendMessage

Step 27：H.handleMessage

Step 28：ActivityThread.handleLaunchActivity

Step 29：ActivityThread.performLaunchActivity

第25步到第29步可以参考7.2节中的第30步到第34步。

Step 30：SubAcitiviyInProcess.onCreate

packages/experimental/Activity/src/shy/luo/activity/SubActivityInProcess.java

```
01 public class SubActivity extends Activity implements OnClickListener {
02     ......
03
04     @Override
05     public void onCreate(Bundle savedInstanceState) {
06         ......
07
08         Log.i(LOG_TAG, "Sub Activity In Process Created.");
09     }
10
11     ......
12 }
```

当SubActivityInProcess组件的成员函数onCreate被调用时，就说明SubActivityInProcess组件被启动起来了。

至此，SubActivityInProcess组件的启动过程就分析完了。与Launcher组件启动MainActivity组件的过程不同，SubActivityInProcess组件与启动它的MainActivity组件是运行在同一个应用程序进程中的。

7.4 子Activity组件在新进程中的启动过程

本节将以应用程序Activity的SubActivityInNewProcess组件的启动过程为例，来分析Android应用程序的子Activity组件在新进程中的启动过程。

虽然MainActivity组件和SubActivityInNewProcess组件都是属于应用程序Activity的，但是当MainActivity组件请求ActivityManagerService启动SubActivityInNewProcess组件时，ActivityManagerService会将SubActivityInNewProcess组件运行在一个新的应用程序进程中。

在前面的7.1节中，当SubActivityInNewProcess组件被MainActivity组件启动起来之后，我们就可以通过adb shell dumpsys activity命令来查看它的信息，如下所示。

```
USER@MACHINE:~/Android$ adb shell dumpsys activity
  ......
  Running activities (most recent first):
    TaskRecord{40739270 #3 A shy.luo.activity}
      Run #2: HistoryRecord{4072e4d0 shy.luo.activity/.SubActivityInNewProcess}
      Run #1: HistoryRecord{40675010 shy.luo.activity/.MainActivity}
    TaskRecord{405c8b98 #2 A com.android.launcher}
      Run #0: HistoryRecord{40554f98 com.android.launcher/com.android.launcher2.Launcher}
  ......
  Running processes (most recent first):
```

```
    Proc # 0: adj=fore /F 406618d8 318:shy.luo.activity.newprocess/10036 (top-activity)
    Proc # 1: adj=bak  /B 4072dc60 309:shy.luo.activity.mainprocess/10036 (bg-activities)
    ......
  ......
```

从Running activities小节的信息可以看出，系统的Activity组件堆栈有TaskRecord#2和TaksRecord#3两个任务。其中，任务TaskRecord#2包含了一个Launcher组件，任务TaksRecord#3包含了一个MainActivity组件和一个SubActivityInNewProcess组件。在任务TaksRecord#3中，由于SubActivityInNewProcess组件是由MainActivity组件启动起来的，因此，前者就位于后者的上面。

由于MainActivity组件和SubActivityInNewProcess组件的android:process属性分别被设置为“shy.luo.activity.mainporcess”和“shy.luo.activity.newprocess”；因此，ActivityManagerService在启动SubActivityInNewProcess组件时，就会发现系统中并不存在一个名称为“shy.luo.activity.newprocess”的应用程序进程。接下来它就会先创建这个应用程序进程，然后再在里面将SubActivityInNewProcess组件启动起来。我们可以用Running processes小节的信息来证实这一点。

SubActivityInNewProcess组件的启动过程与MainActivity组件的启动过程类似，它们都是在一个新的应用程序进程中启动的，唯一的区别是ActivityManagerService不需要为SubActivityInNewProcess组件创建一个新任务；因为它与启动它的MainActivity组件的android:taskAffinity属性都是一样的，即可以将它们放在同一个任务中。在这一点上，SubActivityInNewProcess组件的启动过程与SubActivityInProcess组件的启动过程又是类似的。

结合MainActivity组件和SubActivityInProcess组件的启动过程，我们就可以得到SubActivityInNewProcess组件的启动过程，如下所示。

（1）MainActivity组件向ActivityManagerService发送一个启动SubActivityInNewProcess组件的进程间通信请求。

（2）ActivityManagerService首先将要启动的SubActivityInNewProcess组件的信息保存下来，然后再向MainActivity组件发送一个进入中止状态的进程间通信请求。

（3）MainActivity组件进入到中止状态之后，就会向ActivityManagerService发送一个已进入中止状态的进程间通信请求，以便ActivityManagerService可以继续执行启动SubActivityInNewProcess组件的操作。

（4）ActivityManagerService发现用来运行SubActivityInNewProcess组件的应用程序进程不存在，因此它就会先启动一个新的应用程序进程。

（5）新的应用程序进程启动完成之后，就会向ActivityManagerService发送一个启动完成的进程间通信请求，以便ActivityManagerService可以继续执行启动SubActivityInNewProcess组件的操作。

（6）ActivityManagerService将第2步保存下来的SubActivityInNewProcess组件信息发送给第4步创建的应用程序进程，以便它可以将SubActivityInNewProcess组件启动起来。

由于可以结合7.2小节和7.3节的内容来得到SubActivityInNewProcess组件的启动过程，因此，我们就不具体分析每一个步骤了。

从MainActivity组件、SubActivityInProcess组件和SubActivityInNewProcess组件的启动过程可以看出，一个Activity组件是在一个已经存在的应用程序进程中启动，还是在一个新创建的应用程序进程中启动，是由两个因素决定的。第一个因素是这个Activity组件的用户ID，即PackageManagerService分配给它所属的Android应用程序的用户ID；第二个因素是这个Activity组件的android:process属性。从这一点就可以看出，一个Android应用程序是可以对应于多个应用程序进程的。

从MainActivity组件、SubActivityInProcess组件和SubActivityInNewProcess组件的启动过程还可以看出，每一个Activity组件都是运行在一个任务（Task）中的。任务是一个比进程更高级和抽象的概念，

用来将一系列相关的Activity组件整合在一起，共同完成一个业务功能[1]。由于任务可以将运行在不同进程中的Activity组件整合在一起，因此，这就使得Activity组件的重用性非常强。例如，假设系统中有一个Email应用程序，它里面包含了一个用来发送E-mail的Activity组件EmailSender。在开发其他应用程序时，如果有一个Activity组件SomeActivity，它需要一个E-mail发送功能；那么它就完全可以将Email应用程序中的EmailSender组件启动起来，并且运行在同一个任务中，等到E-mail发送完成之后再返回到SomeActivity组件中。在整个E-mail的发送过程中，用户完全感觉不到SomeActivity组件和EmailSender组件是运行在两个不同的应用程序进程中的。这样，EmailSender组件的重用性就非常强。

1 Activity组件、任务（Task）和进程（Process）的更详细解释,可以参考官方文档：http://developer.android.com/guide/topics/fundamentals/tasks-and-back-stack.html以及http://developer.android.com/guide/topics/fundamentals/processes-and-threads.html。

第 8 章

Service组件的启动过程

Service组件是Android应用程序的四大组件之一，不过与Activity组件不一样，它主要用来处理与用户界面无关的业务逻辑。由于Service组件不直接与用户交互，因此，它涉及的业务逻辑一般都是计算型的，适合在后台运行。

与Activity组件的启动方式类似，Service组件的启动方式也分为显式和隐式两种。对于隐式启动的Service组件来说，我们只需要知道它的组件名称；而对于显式启动的Service组件来说，我们需要知道它的类名称。

Service组件可以被Activity组件启动，也可以被其他的Service组件启动。同时，它既可以在启动它的Activity组件或者Service组件所在的应用程序进程中启动，也可以在一个新的应用程序进程中启动。当一个Service组件被一个Activity组件或者另外一个Service组件启动时，我们可以将它们绑定起来，以便启动者可以方便地得到它的访问接口。

下面我们就结合具体的应用实例来分析Service组件的启动过程。在前面的6.4.2节中，我们开发了一个内存访问服务MemoryService，它是在一个新的应用程序进程中启动的。在接下来的8.1节中，我们再开发一个计数器服务CounterService，它由一个Activity组件启动，并且是在这个Activity组件所在的应用程序进程中启动的。计数器服务CounterService启动起来之后，就会与启动的Activity组件绑定在一起。有了这两个应用实例，在后面的8.2节和8.3节中，我们就分别分析Service组件在新进程中的启动过程以及在进程内的绑定过程。

8.1 Service组件应用实例

在本节中，我们将开发一个名称为Counter的Android应用程序，它由一个Service组件CounterService和一个Activity组件Counter组成。其中，CounterService组件实现了一个计数器服务，它是由Counter组件启动起来的，并且与Counter组件绑定在一起。

CounterService组件在内部使用一个异步任务（AsyncTask）来实现一个计数器。这个计数器每隔1秒就将内部的计数加1，并且实时地将这个计数显示在Counter组件的用户界面上。由于计数器需要不停地执行加数功能，因此，我们将它放在一个后台线程中运行，避免Counter组件不能及时地响应用户界面事件。

应用程序Counter的目录结构如下：

```
~/Android/packages/experimental/Counter
----AndroidManifest.xml
----Android.mk
```

```
----src
    ----shy/luo/counter
        ----ICounterCallback.java
        ----ICounterService.java
        ----CounterService.java
        ----Counter.java
----res
    ----layout
        ----main.xml
    ----values
        ----strings.xml
    ----drawable
        ----icon.png
```

它包含了一个源代码目录src、一个资源目录res、一个配置文件AndroidManifest.xml和一个编译脚本文件Android.mk。接下来，我们就分别介绍每一个文件的内容。

ICounterCallback.java

```
1 package shy.luo.counter;
2
3 public interface ICounterCallback {
4     void count(int val);
5 }
```

这个文件定义了一个计数器回调接口ICounterCallback，它只有一个成员函数count，用来将CounterService组件的当前计数实时地更新到Counter组件的用户界面上。

ICounterService.java

```
1 package shy.luo.counter;
2
3 public interface ICounterService {
4     public void startCounter(int initVal, ICounterCallback callback);
5     public void stopCounter();
6 }
```

这个文件定义了一个计数器接口ICounterService，它有两个成员函数startCounter和stopCounter；其中，前者用来启动计数器，后者用来停止计数器。在启动计数器时，可以指定计数器的初始值，以及更新用户界面的回调接口。

CounterService.java

```
01 package shy.luo.counter;
02
03 import android.app.Service;
04 import android.content.Intent;
05 import android.os.AsyncTask;
06 import android.os.Binder;
07 import android.os.IBinder;
08 import android.util.Log;
09
10 public class CounterService extends Service implements ICounterService {
11     private final static String LOG_TAG = "shy.luo.counter.CounterService";
12
13     private boolean stop = false;
14     private ICounterCallback counterCallback = null;
15
16     private final IBinder binder = new CounterBinder();
17
18     public class CounterBinder extends Binder {
19         public CounterService getService() {
```

```
20              return CounterService.this;
21          }
22      }
23
24      @Override
25      public IBinder onBind(Intent intent) {
26          return binder;
27      }
28
29      @Override
30      public void onCreate() {
31          super.onCreate();
32
33          Log.i(LOG_TAG, "Counter Service Created.");
34      }
35
36      public void startCounter(int initVal, ICounterCallback callback) {
37          counterCallback = callback;
38
39          AsyncTask<Integer, Integer, Integer> task = new AsyncTask<Integer, Integer, Integer>() {
40              @Override
41              protected Integer doInBackground(Integer... vals) {
42                  Integer initCounter = vals[0];
43
44                  stop = false;
45                  while(!stop) {
46                      publishProgress(initCounter);
47
48                      try {
49                          Thread.sleep(1000);
50                      } catch (InterruptedException e) {
51                          e.printStackTrace();
52                      }
53
54                      initCounter++;
55                  }
56
57                  return initCounter;
58              }
59
60              @Override
61              protected void onProgressUpdate(Integer... values) {
62                  super.onProgressUpdate(values);
63
64                  int val = values[0];
65                  counterCallback.count(val);
66              }
67
68              @Override
69              protected void onPostExecute(Integer val) {
70                  counterCallback.count(val);
71              }
72
73          };
74
75          task.execute(initVal);
76      }
77
78      public void stopCounter() {
79          stop = true;
80      }
81  }
```

CounterService是应用程序Counter的一个Service组件，因此，它必须要从Service类继承下来。CounterService组件同时实现了计数器接口ICounterService，用来提供计数器服务。在CounterService组件启动时，它的成员函数onCreate就会被调用；而在CounterService组件被绑定时，它的成员函数onBind就会被调用。

CounterService组件有一个成员变量binder，它指向一个类型为CounterBinder的Binder本地对象。当CounterService组件被绑定时，即它的成员函数onBind被调用时，它就会将内部的Binder本地对象binder返回给绑定者，以便绑定者可以通过这个Binder本地对象来获得它的一个计数器访问接口。

注意

这里之所以要返回一个Binder本地对象给绑定者，是因为绑定者与CounterService组件可能是运行在两个不同的应用程序进程中的。不过，在这个应用实例中，启动CounterService组件的Counter组件与CounterService组件是运行在同一个应用程序进程中的。

CounterService组件的成员函数startCounter和stopCounter分别用来启动和停止内部的计数器；其中，成员函数startCounter使用一个异步任务task来启动一个计数器，成员函数stopCounter通过将CounterService组件的成员变量stop设置为true来停止这个计数器。

当计数器开始执行时，异步任务task的成员函数doInBackground就会被调用。在调用过程中，每隔1秒就会将上一次的计数增加1，并且调用异步任务task的成员函数publishProgress将当前计数发布出去；这会导致异步任务task的成员函数onProgressUpdate被调用，后者主要是使用CounterService组件内部的一个ICounterCallback接口将计数器的当前计数更新到用户界面上。

当计数器被停止执行时，异步任务task的成员函数onPostExecute就会被调用，它主要是使用CounterService组件内部的一个ICounterCallback接口将计数器的最终计数更新到用户界面上。

注意

在异步任务task的三个成员函数中，只有成员函数doInBackground是在后台线程中执行的，其余两个成员函数onProgressUpdate和onPostExecute都是在应用程序Counter的主线程中执行的，因此，它们可以用来更新用户界面[1]。

Counter.java

```
01 package shy.luo.counter;
02
03 import android.app.Activity;
04 import android.content.ComponentName;
05 import android.content.Context;
06 import android.content.Intent;
07 import android.content.IntentFilter;
08 import android.content.ServiceConnection;
09 import android.os.Bundle;
10 import android.os.IBinder;
11 import android.util.Log;
12 import android.view.View;
13 import android.view.View.OnClickListener;
14 import android.widget.Button;
15 import android.widget.TextView;
16
17 public class Counter extends Activity implements OnClickListener, ICounterCallback {
```

1 异步任务（AsyncTask）的具体使用方法可以参考官方文档：http://developer.android.com/reference/android/os/AsyncTask.html。

```
18      private final static String LOG_TAG = "shy.luo.counter.Counter";
19
20      private Button startButton = null;
21      private Button stopButton = null;
22      private TextView counterText = null;
23
24      private ICounterService counterService = null;
25
26      @Override
27      public void onCreate(Bundle savedInstanceState) {
28          super.onCreate(savedInstanceState);
29          setContentView(R.layout.main);
30
31          startButton = (Button)findViewById(R.id.button_start);
32          stopButton = (Button)findViewById(R.id.button_stop);
33          counterText = (TextView)findViewById(R.id.textview_counter);
34
35          startButton.setOnClickListener(this);
36          stopButton.setOnClickListener(this);
37
38          startButton.setEnabled(true);
39          stopButton.setEnabled(false);
40
41          Intent bindIntent = new Intent(Counter.this, CounterService.class);
42          bindService(bindIntent, serviceConnection, Context.BIND_AUTO_CREATE);
43
44          Log.i(LOG_TAG, "Counter Activity Created.");
45      }
46
47
48      @Override
49      public void onDestroy() {
50          super.onDestroy();
51          unbindService(serviceConnection);
52      }
53
54      @Override
55      public void onClick(View v) {
56          if(v.equals(startButton)) {
57              if(counterService != null) {
58                  counterService.startCounter(0, this);
59
60                  startButton.setEnabled(false);
61                  stopButton.setEnabled(true);
62              }
63          } else if(v.equals(stopButton)) {
64              if(counterService != null) {
65                  counterService.stopCounter();
66
67                  startButton.setEnabled(true);
68                  stopButton.setEnabled(false);
69              }
70          }
71      }
72
73      @Override
74      public void count(int val) {
75          String text = String.valueOf(val);
76          counterText.setText(text);
77      }
78
79      private ServiceConnection serviceConnection = new ServiceConnection() {
80          public void onServiceConnected(ComponentName className, IBinder service) {
```

```
            counterService = ((CounterService.CounterBinder)service).getService();

            Log.i(LOG_TAG, "Counter Service Connected");
        }
        public void onServiceDisconnected(ComponentName className) {
            counterService = null;
            Log.i(LOG_TAG, "Counter Service Disconnected");
        }
    };
}
```

Counter是应用程序Counter的根Activity组件，因此，它必须要从类Activity继承下来。Counter同时实现了计数器回调接口ICounterCallback，以便可以实时地更新用户界面上的计数器。

Counter组件是应用程序Counter的主界面，它上面有两个按钮Start Counter和Stop Counter，以及一个文本框。其中，按钮Start Counter和Stop Counter分别用来启动和停止CounterService组件中的计数器，文本框用来实时显示这个计数器的当前计数。

Counter组件被启动时，它的成员函数onCreate 就会被调用，这时候它就会调用其父类ContextWrapper的成员函数bindService来启动CounterService组件。当CounterService组件启动起来之后，Counter组件内部的ServiceConnection对象serviceConnection的成员函数onServiceConnected就会被调用，这时候Counter组件就会获得CounterService组件中的一个Binder本地对象。由于Counter组件和CounterService组件是运行在同一个应用程序进程中的，因此，ServiceConnection对象serviceConnection的成员函数onServiceConnected可以安全地将前面获得的一个Binder本地对象转换为一个类型为CounterBinder的对象，接着再调用它的成员函数getService来获得一个计数器接口。有了这个计数器接口之后，Counter组件就相当于将CounterService组件绑定起来了。

当点击用户界面上的Start Counter按钮时，Counter组件就会调用内部的一个计数器接口counterService的成员函数startCounter来启动CounterService组件中的计数器，并且将自己的一个ICounterCallback接口传递给CounterService组件，以便CounterService组件可以将内部的计数器的当前计数更新到它的用户界面上的文本框中，如它的成员函数count所示。

当点击用户界面上的Stop Counter按钮时，Counter组件就会调用内部的一个计数器接口counterService的成员函数stopCounter来停止CounterService组件中的计数器。

由于Counter组件在启动时绑定了CounterService组件，因此，当它被销毁时，即它的成员函数onDestroy被调用时，它就必须要调用其父类ContextWrapper的成员函数unbindService来解绑CounterService组件。解绑成功之后，Counter组件内部的ServiceConnection对象serviceConnection的成员函数onServiceDisconnected就会被调用，用来执行清理工作。

main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:gravity="center">
    <LinearLayout android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="10px"
        android:orientation="horizontal"
        android:gravity="center">
            <TextView android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_marginRight="4px"
                android:gravity="center"
```

```
16                  android:text="@string/counter">
17              </TextView>
18              <TextView android:id="@+id/textview_counter"
19                  android:layout_width="wrap_content"
20                  android:layout_height="wrap_content"
21                  android:gravity="center"
22                  android:text="@string/init_counter">
23              </TextView>
24      </LinearLayout>
25      <LinearLayout android:layout_width="fill_parent"
26          android:layout_height="wrap_content"
27          android:orientation="horizontal"
28          android:gravity="center">
29              <Button android:id="@+id/button_start"
30                  android:layout_width="wrap_content"
31                  android:layout_height="wrap_content"
32                  android:gravity="center"
33                  android:text="@string/start">
34              </Button>
35              <Button android:id="@+id/button_stop"
36                  android:layout_width="wrap_content"
37                  android:layout_height="wrap_content"
38                  android:gravity="center"
39                  android:text="@string/stop" >
40              </Button>
41          </LinearLayout>
42 </LinearLayout>
```

这是Counter组件的界面配置文件，它在屏幕中间显示两个按钮和一个文本框。

strings.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3      <string name="app_name">Counter</string>
4      <string name="counter">Counter: </string>
5      <string name="init_counter">0</string>
6      <string name="start">Start Counter</string>
7      <string name="stop">Stop Counter</string>
8 </resources>
```

这是应用程序Counter的字符串资源文件，定义了在应用程序中用到的各个字符串。

icon.png

这是应用程序Counter的图标，可以根据需要来放置不同的图片文件。

AndroidManifest.xml

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
03      package="shy.luo.counter"
04      android:versionCode="1"
05      android:versionName="1.0">
06      <application android:icon="@drawable/icon" android:label="@string/app_name">
07           <activity android:name=".Counter"
08               android:label="@string/app_name">
09               <intent-filter>
10                   <action android:name="android.intent.action.MAIN" />
11                   <category android:name="android.intent.category.LAUNCHER" />
12               </intent-filter>
13           </activity>
14           <service android:name=".CounterService"
15               android:enabled="true">
```

```
16          </service>
17      </application>
18 </manifest>
```

这是应用程序Counter的配置文件。由于在程序中使用到了Activity组件Counter和Service组件CounterService，因此，我们就需要在这里对它们进行配置。

注意

由于Counter组件是显式地启动CounterService组件的，因此，这里不需要配置CounterService组件的组件名称。

Android.mk

```
1 LOCAL_PATH:= $(call my-dir)
2 include $(CLEAR_VARS)
3 LOCAL_MODULE_TAGS := optional
4 LOCAL_SRC_FILES := $(call all-subdir-java-files)
5 LOCAL_PACKAGE_NAME := Counter
6 include $(BUILD_PACKAGE)
```

这是应用程序Counter的编译脚本文件。

上述文件都准备好之后，我们就可以对应用程序Counter进行编译和打包了。

```
USER@MACHINE:~/Android$ mmm ./packages/experimental/Counter/
USER@MACHINE:~/Android$ make snod
```

编译成功之后，就可以在out/target/product/gerneric/system/app目录下看到应用程序文件Counter.apk了；打包成功之后，应用程序Counter就包含在out/target/product/gerneric目录下的Android系统镜像文件system.img中了。

最后，我们使用新得到的系统镜像文件system.img来启动Android模拟器。

```
USER@MACHINE:~/Android$ emulator
```

Android模拟器运行起来之后，我们就可以在应用程序启动器中启动应用程序Counter了，即将它的根Activity组件Counter启动起来，它的界面如图8-1所示。

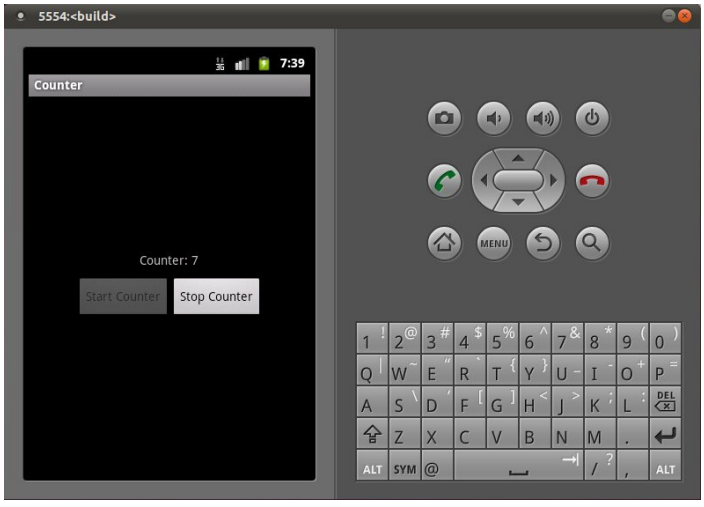


图8-1　应用程序Counter的界面

点击用户界面上的Start Counter按钮，就可以将CounterService组件中的计数器启动起来，并且每隔1秒就可以看到这个计数器的计数增加1；点击用户界面上的Stop Counter按钮，就可以停止

CounterService组件中的计数器。

8.2　Service组件在新进程中的启动过程

本节将以6.4.2节中的应用程序Ashmem的Server组件的启动过程为例，来说明Service组件在新的应用程序进程中的启动过程。

应用程序Ashmem的Server组件是由Client组件启动的。与Activity组件一样，Service组件也是通过Activity管理服务ActivityManagerService启动的。因此，在Server组件的启动过程中，也会涉及Binder进程间通信机制。由于Server组件的android:process属性被设置为“shy.luo.ashmem.server”，因此，它将会在一个名称为“shy.luo.ashmem.server”的应用程序进程中启动。

Client组件启动Server组件的过程如下所示。

（1）Client组件向ActivityManagerService发送一个启动Server组件的进程间通信请求。

（2）ActivityManagerService发现用来运行Server组件的应用程序进程不存在，因此，它就会首先将Server组件的信息保存下来，接着再创建一个新的应用程序进程。

（3）新的应用程序进程启动完成之后，就会向ActivityManagerService发送一个启动完成的进程间通信请求，以便ActivityManagerService可以继续执行启动Server组件的操作。

（4）ActivityManagerService将第2步保存下来的Server组件信息发送给第2步创建的应用程序进程，以便它可以将Server组件启动起来。

这个过程一共分为19个步骤，接下来我们就详细分析每一个步骤。

第1步到第4步如图8-2所示。

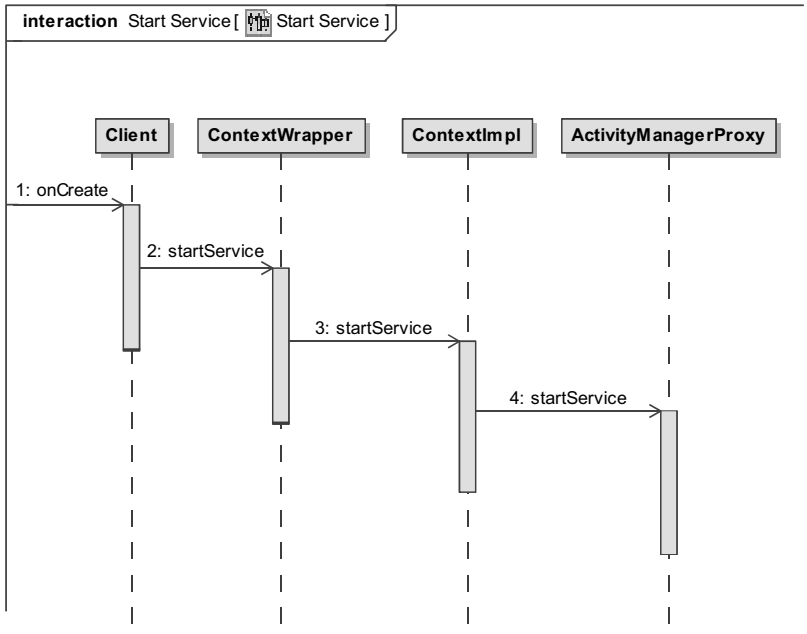


图8-2　Server组件的启动过程——Step 1至Step 4

Step 1：Client.onCreate

packages/experimental/Ashmem/src/shy/luo/ashmem/Client.java

```
01 public class Client extends Activity implements OnClickListener {
02         ......
```

```
03
04          @Override
05          public void onCreate(Bundle savedInstanceState) {
06                  ......
07
08                  IMemoryService ms = getMemoryService();
09                  if(ms == null) {
10                          Log.i(LOG_TAG, "start shy.luo.ashmem.server");
11                          startService(new Intent("shy.luo.ashmem.server"));
12                  } else {
13                          Log.i(LOG_TAG, "Memory Service has started.");
14                  }
15
16                  ......
17          }
18
19           ......
20  }
```

第8行代码首先调用成员函数getMemoryService来获得内存访问服务MemoryService的一个代理对象。如果获得的代理对象为null，那么就说明内存访问服务MemoryService尚未启动起来，因此，接下来第11行就调用父类ContextWrapper的成员函数startService将应用程序Ashmem的Server组件启动起来，以便它可以将其内部的内存访问服务MemoryService启动起来。

由于应用程序Ashmem的Server组件的组件名称被设置为“shy.luo.ashmem.server”，因此，第11行在调用父类ContextWrapper的成员函数startService启动它时，只要将参数intent的Action名称指定为“shy.luo.ashmem.server”即可。

Step 2：ContextWrapper.startService

frameworks/base/core/java/android/content/ContextWrapper.java

```
01 public class ContextWrapper extends Context {
02     Context mBase;
03     ......
04
05     @Override
06     public ComponentName startService(Intent service) {
07         return mBase.startService(service);
08     }
09
10     ......
11 }
```

ContextWrapper类有一个类型为Context的成员变量mBase，它实际上指向的是一个ContextImpl对象。由前面7.2小节中的Step 34可以知道，每一个Activity组件在启动时，ActivityThread类都会为它创建一个ContextImpl对象，用来描述它的运行上下文环境。这个ContextImpl对象就保存在这个Activity组件的父类ContextWrapper的成员变量mBase中。

因此，第7行代码实际上是调用了ContextImpl类的成员函数startService来执行启动Server组件的操作。

Step 3：ContextImpl. startService

frameworks/base/core/java/android/app/ContextImpl.java

```
01 class ContextImpl extends Context {
02     ......
03
04     @Override
05     public ComponentName startService(Intent service) {
06         try {
07             ComponentName cn = ActivityManagerNative.getDefault().startService(
```

```
08                  mMainThread.getApplicationThread(), service,
09                  service.resolveTypeIfNeeded(getContentResolver()));
10              ......
11              return cn;
12          } catch (RemoteException e) {
13              return null;
14          }
15      }
16
17      ......
18  }
```

第7行到第9行代码首先通过调用ActivityManagerNative类的静态成员函数getDefault来获得ActivityManagerService的一个代理对象，接着再调用这个代理对象的成员函数startService来请求ActivityManagerService启动Server组件。

在前面的7.2小节中提到，ContextImpl类的成员变量mMainThread的类型为ActivityThread，调用它的成员函数getApplicationThread就可以获得当前应用程序进程中一个类型为ApplicationThread的Binder本地对象。第7行到第9行代码需要将这个ApplicationThread对象传递给ActivityManagerService，以便ActivityManagerService可以知道是谁请求它启动Server组件。

ActivityManagerService代理对象的类型为ActivityManagerProxy，因此，接下来就会调用ActivityManagerProxy类的成员函数startService向ActivityManagerService发送一个进程间通信请求。

Step 4：ActivityManagerProxy. startService

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
01 class ActivityManagerProxy implements IActivityManager
02 {
03     ......
04
05     public ComponentName startService(IApplicationThread caller, Intent service,
06             String resolvedType) throws RemoteException
07     {
08         Parcel data = Parcel.obtain();
09         Parcel reply = Parcel.obtain();
10         data.writeInterfaceToken(IActivityManager.descriptor);
11         data.writeStrongBinder(caller != null ? caller.asBinder() : null);
12         service.writeToParcel(data, 0);
13         data.writeString(resolvedType);
14         mRemote.transact(START_SERVICE_TRANSACTION, data, reply, 0);
15         reply.readException();
16         ComponentName res = ComponentName.readFromParcel(reply);
17         data.recycle();
18         reply.recycle();
19         return res;
20     }
21
22     ......
23 }
```

第10行到第13行代码将前面传进来的参数写入到Parcel对象data中，接着再通过ActivityManagerProxy类内部的一个Binder代理对象mRemote向ActivityManagerService发送一个类型为START_SERVICE_TRANSACTION的进程间通信请求。

以上4步都是在应用程序Ashmem中执行的，接下来第5步到第8步是在ActivityManagerService中执行的，主要是用来处理Client组件发出的类型为START_SERVICE_TRANSACTION的进程间通信请求，如图8–3所示。

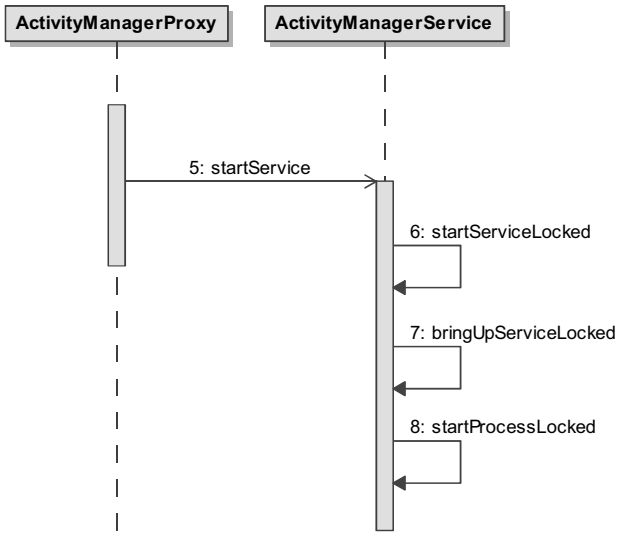


图8-3　Server组件的启动过程——Step 5至Step 8

Step 5：ActivityManagerService.startService

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02                          implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     public ComponentName startService(IApplicationThread caller, Intent service,
06             String resolvedType) {
07           ......
08
09           synchronized(this) {
10               final int callingPid = Binder.getCallingPid();
11               final int callingUid = Binder.getCallingUid();
12               final long origId = Binder.clearCallingIdentity();
13               ComponentName res = startServiceLocked(caller, service,
14                   resolvedType, callingPid, callingUid);
15               Binder.restoreCallingIdentity(origId);
16               return res;
17           }
18     }
19
20     ......
21 }
```

ActivityManagerService类的成员函数startService用来处理类型为START_SERVICE_TRANSACTION的进程间通信请求。

第13行和第14行代码调用ActivityManagerService类的另外一个成员函数startServiceLocked 来进一步处理类型为START_SERVICE_TRANSACTION的进程间通信请求，即执行一个启动Server组件的操作。

Step 6：ActivityManagerService.startServiceLocked

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02                          implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     ComponentName startServiceLocked(IApplicationThread caller,
```

```
06              Intent service, String resolvedType,
07              int callingPid, int callingUid) {
08          synchronized(this) {
09              ......
10
11              ServiceLookupResult res =
12                  retrieveServiceLocked(service, resolvedType,
13                  callingPid, callingUid);
14              ......
15
16              ServiceRecord r = res.record;
17              ......
18
19              if (!bringUpServiceLocked(r, service.getFlags(), false)) {
20                  return new ComponentName("!", "Service process is bad");
21              }
22              return r.name;
23          }
24      }
25
26      ......
27  }
```

在ActivityManagerService中，每一个Service组件都使用一个ServiceRecord对象来描述，就像每一个Activity组件都使用一个ActivityRecord对象来描述一样。

第11行到第13行代码首先调用成员函数retrieveServiceLocked在ActivityManagerService中查找是否存在与参数service对应的一个ServiceRecord对象。如果不存在，那么ActivityManagerService就会到PackageManagerService中去获取与参数service对应的一个Service组件的信息，然后将这些信息封装成一个ServiceRecord对象，最后将这个ServiceRecord对象封装成一个ServiceLookupResult对象返回给调用者。

第16行代码获得了与参数service对应的一个ServiceRecord对象r之后，接着第19行就调用ActivityManagerService类的另外一个成员函数bringUpServiceLocked来启动ServiceRecord对象r所描述的一个Service组件，即Server组件。

Step 7：ActivityManagerService.bringUpServiceLocked

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02                              implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03      ......
04
05      private final boolean bringUpServiceLocked(ServiceRecord r,
06                      int intentFlags, boolean whileRestarting) {
07          ......
08
09          final String appName = r.processName;
10          ProcessRecord app = getProcessRecordLocked(appName, r.appInfo.uid);
11          ......
12
13          if (app != null && app.thread != null) {
14              try {
15                  realStartServiceLocked(r, app);
16                  return true;
17              } catch (RemoteException e) {
18                  ......
19              }
20          }
21
22          // Not running -- get it started, and enqueue this service record
```

```
23          // to be executed when the app comes up.
24          if (startProcessLocked(appName, r.appInfo, true, intentFlags,
25                      "service", r.name, false) == null) {
26              ......
27              return false;
28          }
29
30          if (!mPendingServices.contains(r)) {
31              mPendingServices.add(r);
32          }
33
34          return true;
35      }
36
37      ......
38  }
```

第9行代码首先获得ServiceRecord对象r所描述的Service组件的android:process属性，并且保存在变量appName中。接着第10行根据这个属性的值，以及ServiceRecord对象r所描述的Service组件的用户ID在ActivityManagerService中查找是否已经存在一个对应的ProcessRecord对象app。如果存在，那么就说明用来运行这个Service组件的应用程序进程已经存在了，因此，第15行就会直接调用成员函数realStartServiceLocked在ProcessRecord对象app所描述的应用程序进程中启动这个Service组件。

由于ServiceRecord对象r所描述的Service组件的android:process属性被设置为“shy.luo.ashmem.server”，即它要求运行在一个独立的应用程序进程中，因此，我们假设这个应用程序进程尚未创建，即第13行的if语句为false，接下来第24行就会调用成员函数startProcessLocked来启动一个名称为“shy.luo.ashmem.server”的应用程序进程。

最后，第31行代码将ServiceRecord对象r保存在ActivityManagerService类的成员变量mPendingService中，表示它描述的一个Service组件正在等待启动。

Step 8：ActivityManagerService.startProcessLocked

第8步可以参考7.2小节中的第23步，它主要是调用Process类的静态成员函数start来创建一个新的应用程序进程，这个新创建的应用程序进程以ActivityThread类的静态成员函数main为入口。

以上4步都是在ActivityManagerService中执行的，接下来第9步和第10步是在新创建的应用程序进程中执行的，主要是用来初始化新创建的应用程序进程，如图8–4所示。

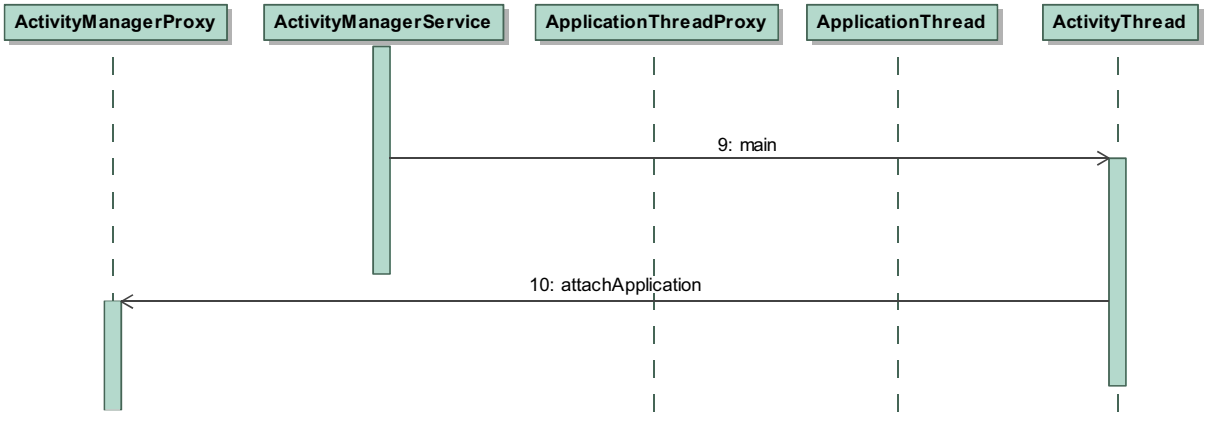


图8–4　Server组件的启动过程——Step 9和Step 10

Step 9：ActivityThread.main

Step 10：ActivityManagerProxy.attachApplication

第9步和第10步可以参考7.2小节中的第24步和第25步，其中，第9步会在新创建的应用程序进程中创建一个ActivityThread对象和一个ApplicationThread对象；第10步会向ActivityManagerService发出一个类型为ATTACH_APPLICATION_TRANSACTION的进程间通信请求，以及将前面创建的一个ApplicationThread对象传递给ActivityManagerService，以便ActivityManagerService可以和这个新创建的应用程序进程执行Binder进程间通信。

以上两步都是在新创建的应用程序进程中执行的，接下来第11步到第14步是在ActivityManagerService中执行的，主要是用来处理新的应用程序进程发出的类型为ATTACH_APPLICATION_TRANSACTION的进程间通信请求，如图8-5所示。

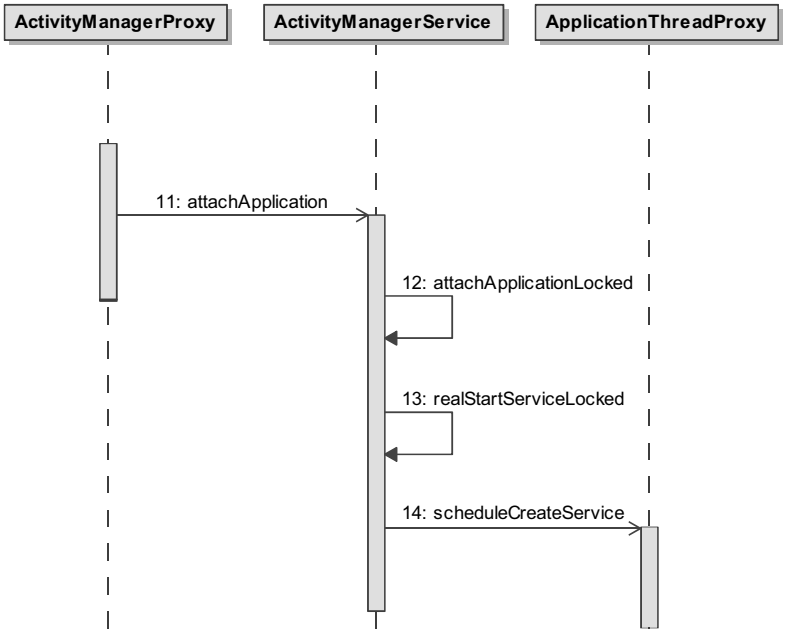


图8-5　Server组件的启动过程——Step 11至Step 14

Step 11：ActivityManagerService.attachApplication

第11 步可以参考7.2小节中的第26步，它调用了ActivityManagerService类的成员函数attachApplicationLocked来处理类型为ATTACH_APPLICATION_TRANSACTION的进程间通信请求，用来执行启动Server组件的操作。

Step 12：ActivityManagerService.attachApplicationLocked

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02                          implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     private final boolean attachApplicationLocked(IApplicationThread thread,
06             int pid) {
07         // Find the application record that is being attached...  either via
08         // the pid if we are running in multiple processes, or just pull the
09         // next app record if we are emulating process with anonymous threads.
10         ProcessRecord app;
11         if (pid != MY_PID && pid >= 0) {
```

```
12              synchronized (mPidsSelfLocked) {
13                  app = mPidsSelfLocked.get(pid);
14              }
15          }
16          ......
17
18          String processName = app.processName;
19          ......
20
21          app.thread = thread;
22          ......
23
24          boolean badApp = false;
25          ......
26
27          // See if the top visible activity is waiting to run in this process...
28          ActivityRecord hr = mMainStack.topRunningActivityLocked(null);
29          if (hr != null && normalMode) {
30              if (hr.app == null && app.info.uid == hr.info.applicationInfo.uid
31                      && processName.equals(hr.processName)) {
32                  try {
33                      if (mMainStack.realStartActivityLocked(hr, app, true, true)) {
34                          ......
35                      }
36                  } catch (Exception e) {
37                      ......
38                      badApp = true;
39                  }
40              } else {
41                  mMainStack.ensureActivitiesVisibleLocked(hr, null, processName, 0);
42              }
43          }
44
45          // Find any services that should be running in this process...
46          if (!badApp && mPendingServices.size() > 0) {
47              ServiceRecord sr = null;
48              try {
49                  for (int i=0; i<mPendingServices.size(); i++) {
50                      sr = mPendingServices.get(i);
51                      if (app.info.uid != sr.appInfo.uid
52                          || !processName.equals(sr.processName)) {
53                              continue;
54                      }
55
56                      mPendingServices.remove(i);
57                      i--;
58                      realStartServiceLocked(sr, app);
59                      ......
60                  }
61              } catch (Exception e) {
62                  ......
63              }
64          }
65
66          ......
67          return true;
68      }
69
70      ......
71  }
```

参数pid指向的是前面所创建的应用程序进程的PID。在前面的Step 8中，ActivityManagerService以这个PID为关键字将一个ProcessRecord对象保存在了成员变量mPidsSelfLocked中，因此，第10行到第

15行代码就可以通过参数pid将这个ProcessRecord对象取回来，并且保存在变量app中。

前面得到的ProcessRecord对象app就是用来描述新创建的应用程序进程的，现在既然这个应用程序进程已经启动起来了，第21行代码就将它的成员变量thread设置为参数thread所指向的一个ApplicationThread代理对象，这样，ActivityManagerService以后就可以通过这个ApplicationThread代理对象来和新创建的应用程序进程进行通信了。

第28行到第43行代码检查位于Activity组件堆栈顶端的Activity组件是否需要在新创建的应用程序进程中启动。如果需要，那么第33行代码就会调用成员变量mMainStack的成员函数realStartActivityLocked将这个Activity组件启动起来。如果位于Activity组件堆栈顶端的Activity组件已经在新创建的应用程序进程中启动起来了，但是它的窗口没有显示出来，那么第41行代码就会调用成员变量mMainStack的成员函数ensureActivitiesVisibleLocked把它的窗口显示出来。从这里就可以看出，当一个应用程序进程启动完成之后，ActivityManagerService会优先处理需要在它里面启动或者显示的Activity组件，然后再处理需要在它里面启动的Service组件，这是因为Activity组件是需要与用户交互的，而Service组件则不需要。

第46行到第64行代码检查保存在成员变量mPendingServices中的Service组件是否需要在新创建的应用程序进程中启动。如果需要，那么第56行代码就首先将它从成员变量mPendingServices中删除，接着第58行再调用成员函数realStartServiceLocked将它启动起来。

在前面的Step 7中，我们将一个ServiceRecord对象保存在了成员变量mPendingServices中，这个ServiceRecord对象描述的是应用程序Ashmem中的Server组件，它正在等待在ProcessRecord对象app所描述的应用程序进程中启动。因此，接下来就会调用ActivityManagerService类的成员函数realStartServiceLocked来启动它。

Step 13：ActivityManagerService.realStartServiceLocked

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 class ActivityManagerProxy implements IActivityManager
02 {
03     ......
04
05     private final void realStartServiceLocked(ServiceRecord r,
06             ProcessRecord app) throws RemoteException {
07         ......
08
09         r.app = app;
10         ......
11
12         try {
13             ......
14
15             app.thread.scheduleCreateService(r, r.serviceInfo);
16
17             ......
18         } finally {
19             ......
20         }
21
22         ......
23     }
24
25     ......
26 }
```

第9行代码将ServiceRecord对象r的成员变量app设置为ProcessRecord对象app，表示ServiceRecord对象r所描述的Service组件是在ProcessRecord对象app所描述的应用程序进程中启动的。

ProcessRecord对象app的成员变量thread是一个类型为ApplicationThreadProxy的Binder代理对象，它

指向了新创建的应用程序进程中的一个ApplicationThread对象。因此，第15行就可以调用它的成员函数scheduleCreateService来请求新创建的应用程序进程将ServiceRecord对象r所描述的Service组件启动起来。

Step 14：ApplicationThreadProxy.scheduleCreateService

frameworks/base/core/java/android/app/ApplicationThreadNative.java

```
01 class ApplicationThreadProxy implements IApplicationThread {
02     ......
03
04     public final void scheduleCreateService(IBinder token, ServiceInfo info)
05                 throws RemoteException {
06         Parcel data = Parcel.obtain();
07         data.writeInterfaceToken(IApplicationThread.descriptor);
08         data.writeStrongBinder(token);
09         info.writeToParcel(data, 0);
10         mRemote.transact(SCHEDULE_CREATE_SERVICE_TRANSACTION, data, null,
11             IBinder.FLAG_ONEWAY);
12         data.recycle();
13     }
14
15     ......
16 }
```

第7行到第9行代码将前面传进来的参数写入到Parcel对象data中，接着再通过ApplicationThreadProxy类内部的一个Binder代理对象mRemote向新创建的应用程序进程发送一个类型为SCHEDULE_CREATE_SERVICE_TRANSACTION的进程间通信请求。

以上4步都是在ActivityManagerService中执行的，接下来第15步到第19步是在新创建的应用程序进程中执行的，主要是用来处理ActivityManagerService发出的类型为SCHEDULE_CREATE_SERVICE_TRANSACTION的进程间通信请求，如图8-6所示。

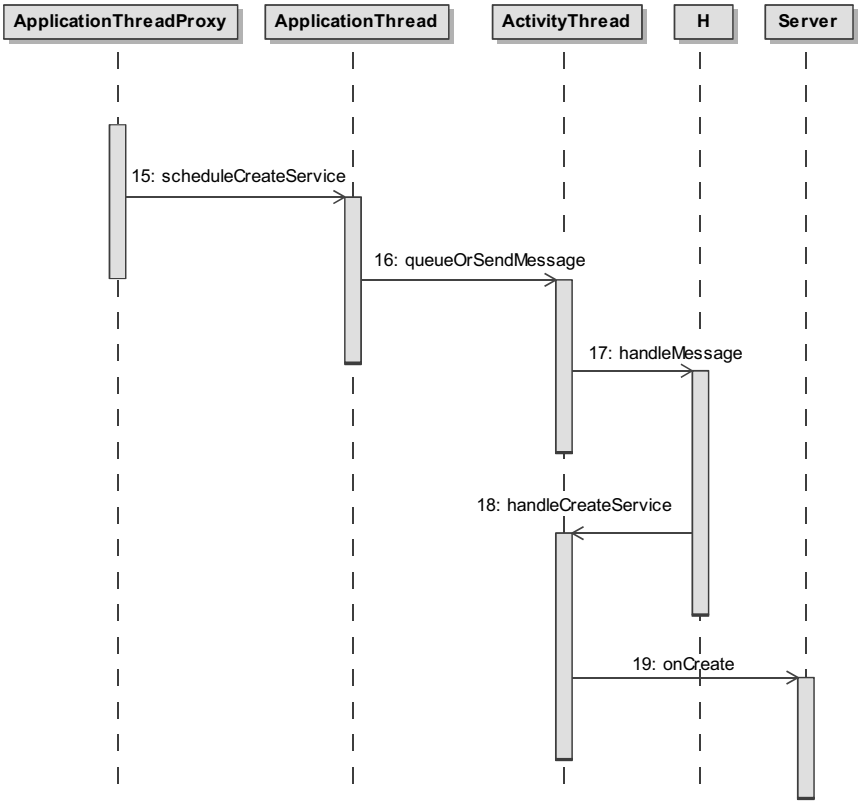


图8-6　Server组件的启动过程——Step 15至Step 19

Step 15：ApplicationThread.scheduleCreateService

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final class ApplicationThread extends ApplicationThreadNative {
05         ......
06
07         public final void scheduleCreateService(IBinder token,
08         ServiceInfo info) {
09             CreateServiceData s = new CreateServiceData();
10             s.token = token;
11             s.info = info;
12
13             queueOrSendMessage(H.CREATE_SERVICE, s);
14         }
15
16         ......
17     }
18
19     ......
20 }
```

ApplicationThread类的成员函数scheduleCreateService用来处理类型为SCHEDULE_CREATE_SERVICE_TRANSACTION的进程间通信请求。

第9行到第11行代码首先将要启动的Service组件的信息封装成一个CreateServiceData对象，接着再调用ActivityThread类的成员函数queueOrSendMessage将这个CreateServiceData对象封装成一个类型为CREATE_SERVICE的消息，以便可以将它发送到新创建的应用程序进程的主线程的消息队列中。

Step 16：ActivityThread.queueOrSendMessage

第16步可以参考7.2小节中的第31步，它主要是将上一步创建的CreateServiceData对象封装成一个类型为CREATE_SERVICE的消息，然后调用ActivityThread类的成员变量mH的成员函数sendMessage将这个消息发送到新创建的应用程序进程的主线程的消息队列中。这个消息最终是在ActivityThread类的成员变量mH的成员函数handleMessage中处理的。

Step 17：H.handleMessage

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final class H extends Handler {
05         ......
06
07         public void handleMessage(Message msg) {
08             ......
09
10             switch (msg.what) {
11                 ......
12                 case CREATE_SERVICE:
13                     handleCreateService((CreateServiceData)msg.obj);
14                     break;
15                 ......
16             }
17
18             ......
19         }
```

```
20
21          ......
22      }
23
24      ......
25  }
```

第13行的代码首先将Message对象msg的成员变量obj转换成一个CreateServiceData对象，它里面包含了要启动的Service组件的信息，接着再调用ActivityThread类的成员函数handleCreateService将这个Service组件启动起来。

Step 18：ActivityThread.handleCreateService

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02      ......
03
04      private final void handleCreateService(CreateServiceData data) {
05          ......
06
07          LoadedApk packageInfo = getPackageInfoNoCheck(
08              data.info.applicationInfo);
09          Service service = null;
10          try {
11              java.lang.ClassLoader cl = packageInfo.getClassLoader();
12              service = (Service) cl.loadClass(data.info.name).newInstance();
13          } catch (Exception e) {
14              ......
15          }
16
17          try {
18              ......
19
20              ContextImpl context = new ContextImpl();
21              context.init(packageInfo, null, this);
22
23              Application app = packageInfo.makeApplication(false, mInstrumentation);
24              context.setOuterContext(service);
25              service.attach(context, this, data.info.name, data.token, app,
26                  ActivityManagerNative.getDefault());
27              service.onCreate();
28              mServices.put(data.token, service);
29              ......
30          } catch (Exception e) {
31              ......
32          }
33      }
34
35      ......
36  }
```

第7行调用成员函数getPackageInfoNoCheck来获得一个用来描述即将要启动的Service组件所在的应用程序的LoadedApk对象，并且将它保存在变量packageInfo中。在前面7.2小节的Step 32中提到，在进程中加载的每一个应用程序都使用一个LoadedApk对象来描述，通过它就可以访问到它所描述的应用程序的资源。

第11行调用LoadedApk对象packageInfo的成员函数getClassLoader来获得一个类加载器，接着第12行通过这个类加载器将CreateServiceData对象data所描述的一个Service组件加载到内存中，并且创建它的一个实例，保存在Service对象service中。由于CreateServiceData对象data所描述的Service组件即为应用程序Ashmem中的Server组件，因此，Service对象service指向的Service组件实际上是一个Server组件。

第20行和第21行代码创建和初始化了一个ContextImpl对象context，用来作为前面所创建的Service对象service的运行上下文环境，通过它就可以访问到特定的应用程序资源，以及启动其他的应用程序组件。

第23行创建了一个Application对象app，用来描述Service对象service所属的应用程序。接着第25行代码使用Application对象app、ContextImpl对象context和CreateServiceData对象data来初始化Service对象service。

Service对象service初始化完成之后，第27行就会调用它的成员函数onCreate，以便让它执行一些自定义的初始化工作。一般来说，我们在自定义一个Service组件时，都会重写其父类Service的成员函数onCreate，以便可以执行一些业务相关的初始化工作。

最后，第28行以CreateServiceData对象data的成员变量token为关键字，将Service对象service保存在ActivityThread类的成员变量mServices中。

注意

CreateServiceData对象data的成员变量token是一个Binder代理对象，它指向了ActivityManagerService内部的一个ServiceRecord对象，用来描述前面已经启动起来的Service组件。

由于Service对象service指向的是一个Server组件，因此，启动Server组件的最后一步便是调用它的成员函数onCreate，以便让它有机会执行自定义的初始化工作。

Step 19：Server.onCreate

packages/experimental/Ashmem/src/shy/luo/ashmem/Server.java

```
01 public class Server extends Service {
02     ......
03
04     @Override
05     public void onCreate() {
06         memoryService = new MemoryService();
07
08         try {
09             ServiceManager.addService("AnonymousSharedMemory", memoryService);
10             Log.i(LOG_TAG, "Succeed to add memory service.");
11         } catch (RuntimeException ex) {
12             Log.i(LOG_TAG, "Failed to add Memory Service.");
13             ex.printStackTrace();
14         }
15     }
16
17     ......
18 }
```

Server组件的成员函数onCreate主要用来启动一个内存访问服务MemoryService，并且将它注册到Service Manager中。

至此，Server组件的启动过程就分析完了，它是在一个新的应用程序进程中启动的。接下来，我们继续分析Service组件在启动它的组件所在的应用程序进程内的绑定过程。

8.3　Service组件在进程内的绑定过程

本节将以8.1小节中应用程序Counter的CounterService组件的绑定过程为例，来说明Service组件在启动它的组件所在的应用程序进程内的绑定过程。

在应用程序Counter中，CounterService组件是被Counter组件绑定起来的。Counter组件在绑定CounterService组件之前，首先会将它启动起来。CounterService组件的启动过程与前面8.2小节所介绍的Server组件的启动过程类似，只不过前者是在已有的一个应用程序进程中启动的，而后者是在一个新的应用程序进程中启动的。

Counter组件绑定CounterService组件的过程如下所示。

（1）Counter组件向ActivityManagerService发送一个绑定CounterService组件的进程间通信请求。

（2）ActivityManagerService发现用来运行CounterService组件的应用程序进程即为Counter组件所运行在的应用程序进程，因此，它就直接通知该应用程序进程将CounterService组件启动起来。

（3）CounterService组件启动起来之后，ActivityManagerService就请求它返回一个Binder本地对象，以便Counter组件可以通过这个Binder本地对象来和CounterService组件建立连接。

（4）ActivityManagerService将前面从CounterService组件中获得的一个Binder本地对象发送给Counter组件。

（5）Counter组件获得了ActivityManagerService给它发送的Binder本地对象之后，就可以通过它来获得CounterService组件的一个访问接口。Counter组件以后就可以通过这个访问接口来使用CounterService组件所提供的服务，这就相当于将CounterService组件绑定在Counter组件内部了。

这个过程一共分为27个步骤，接下来我们就详细分析每一个步骤。

第1步到第4步如图8-7所示。

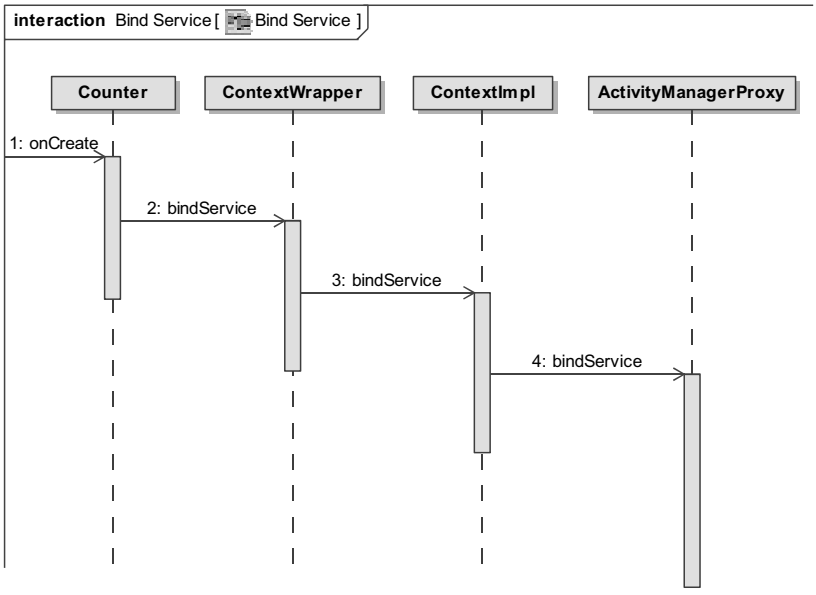


图8-7　CounterService组件的绑定过程——Step 1至Step 4

Step 1：Counter.onCreate

packages/experimental/Counter/src/shy/luo/counter/Counter.java

```
01 public class Counter extends Activity implements OnClickListener, ICounterCallback {
02     ......
03
04     @Override
05     public void onCreate(Bundle savedInstanceState) {
06         .....
07
08         Intent bindIntent = new Intent(Counter.this, CounterService.class);
09         bindService(bindIntent, serviceConnection, Context.BIND_AUTO_CREATE);
10
```

```
11          Log.i(LOG_TAG, "Counter Activity Created.");
12      }
13
14      ......
15
16      private ServiceConnection serviceConnection = new ServiceConnection() {
17          public void onServiceConnected(ComponentName className, IBinder service) {
18              counterService = ((CounterService.CounterBinder)service).getService();
19
20              Log.i(LOG_TAG, "Counter Service Connected");
21          }
22          .....
23      };
24  }
```

第8行代码以CounterService组件的类名来创建一个Intent对象，接着第9行再以这个Intent对象为参数来调用父类ContextWrapper的成员函数bindService，以便可以执行绑定CounterService组件的操作。

注意

Counter组件在绑定CounterService组件时，还指定了另外一个参数serviceConnection，它是Counter组件的一个成员变量，类型为ServiceConnection。当Counter组件成功地绑定了CounterService组件之后，Counter组件的成员变量serviceConnection的成员函数onServiceConnected就会被调用，以便Counter组件可以获得CounterService组件的访问接口。

Step 2：ContextWrapper.bindService

frameworks/base/core/java/android/content/ContextWrapper.java

```
01 public class ContextWrapper extends Context {
02      Context mBase;
03      ......
04
05      @Override
06      public boolean bindService(Intent service, ServiceConnection conn,
07              int flags) {
08          return mBase.bindService(service, conn, flags);
09      }
10
11      ......
12  }
```

在前面8.2小节的Step 2中提到，ContextWrapper类的成员变量mBase指向的是一个ContextImpl对象，因此，第8行代码实际上是调用了ContextImpl类的成员函数bindService来执行绑定CounterService组件的操作。

Step 3：ContextImpl.bindService

frameworks/base/core/java/android/app/ContextImpl.java

```
01 class ContextImpl extends Context {
02      ......
03
04      /*package*/ LoadedApk mPackageInfo;
05      /*package*/ ActivityThread mMainThread;
06      private Context mOuterContext;
07      ......
08
09      @Override
10      public boolean bindService(Intent service, ServiceConnection conn,
11              int flags) {
12          IServiceConnection sd;
13          if (mPackageInfo != null) {
```

```
14              sd = mPackageInfo.getServiceDispatcher(conn, getOuterContext(),
15                  mMainThread.getHandler(), flags);
16          } else {
17              ......
18          }
19          try {
20              int res = ActivityManagerNative.getDefault().bindService(
21                  mMainThread.getApplicationThread(), getActivityToken(),
22                  service, service.resolveTypeIfNeeded(getContentResolver()),
23                  sd, flags);
24              ......
25              return res != 0;
26          } catch (RemoteException e) {
27              return false;
28          }
29      }
30
31      ......
32  }
```

ContextImpl类的成员变量mPackageInfo的类型为LoadedApk，第14行和第15行代码调用它的成员函数getServiceDispatcher将ServiceConnection对象conn封装成一个实现了IServiceConnection接口的Binder本地对象sd，接着第20行到第23行代码再调用ActivityManagerService代理对象的成员函数bindService将前面获得的Binder本地对象sd，以及Intent对象service等信息发送给ActivityManagerService，以便ActivityManagerService可以将CounterService组件启动起来，并且将它连接到Counter组件中。

在将ServiceConnection 对象conn封装成一个实现了IServiceConnection接口的Binder本地对象sd的过程中，还使用到了另外两个参数。

第一个参数是通过调用成员变量mMainThread 的成员函数getHandler来获得的。成员变量mMainThread 的类型为ActivityThread，它的成员函数getHandler返回的是其内部的一个成员变量mH。在前面7.2小节的Step 14中提到，ActivityThread类的成员变量mH是用来向当前应用程序进程的主线程的消息队列发送消息的。当ActivityManagerService成功地将CounterService组件启动起来，并且获得了它内部的一个Binder本地对象之后，ActivityManagerService便会将这个Binder本地对象传递给Binder本地对象sd，接着Binder本地对象sd再将这个Binder本地对象封装成一个消息，并且发送到Counter组件所运行在的应用程序进程的主线程的消息队列中，最后再分发给Counter组件内部的成员变量serviceConnection的成员函数onServiceConnected来处理。

第二个参数是通过调用成员函数getOuterContext来获得的。ContextImpl类的成员函数getOuterContext返回的是其内部的一个成员变量mOuterContext。ContextImpl类的成员变量mOuterContext的类型为Context，它指向的是一个Activity组件。在我们这个场景中，这个Activity组件便是Counter组件了。这样，前面所获得的Binder本地对象sd就可以知道它所封装的ServiceConnection 对象conn是与Counter组件关联在一起的。

接下来，我们继续分析ContextImpl类的成员变量mPackageInfo的成员函数getServiceDispatcher是如何将ServiceConnection 对象conn封装成一个实现了IServiceConnection接口的Binder本地对象sd的，它的实现如下所示。

frameworks/base/core/java/android/app/LoadedApk.java

```
01 final class LoadedApk {
02     ......
03
04     public final IServiceConnection getServiceDispatcher(ServiceConnection c,
05             Context context, Handler handler, int flags) {
06         synchronized (mServices) {
07             LoadedApk.ServiceDispatcher sd = null;
```

```
08              HashMap<ServiceConnection, LoadedApk.ServiceDispatcher> map = mServices.get(context);
09              if (map != null) {
10                  sd = map.get(c);
11              }
12              if (sd == null) {
13                  sd = new ServiceDispatcher(c, context, handler, flags);
14                  if (map == null) {
15                      map = new HashMap<ServiceConnection, LoadedApk.ServiceDispatcher>();
16                      mServices.put(context, map);
17                  }
18                  map.put(c, sd);
19              } else {
20                  sd.validate(context, handler);
21              }
22              return sd.getIServiceConnection();
23          }
24      }
25
26      ......
27  }
```

每一个绑定过Service组件的Activity组件在LoadedApk类中都有一个对应的ServiceDispatcher对象，它负责将这个被绑定的Service组件与绑定它的Activity组件关联在一起。这些ServiceDispatcher对象保存在一个HashMap中，并且以它们所关联的ServiceConnection对象为关键字。最后，用来保存这些ServiceDispatcher对象的HashMap又以它们所关联的Activity组件的Context接口为关键字保存在LoadedApk类的成员变量mServices中。

第7行到第21行代码在LoadedApk类的成员变量mServices中检查是否存在一个以ServiceConnection对象c为关键字的ServiceDispatcher对象sd。如果不存在，那么第13行代码就会首先创建这个ServiceDispatcher对象sd，接着第18行代码再将它保存在以Context对象context为关键字的HashMap对象map中。如果以Context对象context为关键字的HashMap对象map不存在，那么第15行代码就会先创建它，接着第16行代码再将它保存在LoadedApk类的成员变量mServices中。

最后，第22行代码调用前面所获得的ServiceDispatcher对象sd的成员函数getIServiceConnection来获得一个实现了IServiceConnection接口的Binder本地对象，它的实现如下所示。

frameworks/base/core/java/android/app/LoadedApk.java

```
01 final class LoadedApk {
02      ......
03
04      static final class ServiceDispatcher {
05          private final ServiceDispatcher.InnerConnection mIServiceConnection;
06          private final ServiceConnection mConnection;
07          private final Handler mActivityThread;
08          private final Context mContext;
09          ......
10
11          private static class InnerConnection extends IServiceConnection.Stub {
12              final WeakReference<LoadedApk.ServiceDispatcher> mDispatcher;
13              ......
14
15              InnerConnection(LoadedApk.ServiceDispatcher sd) {
16                  mDispatcher = new WeakReference<LoadedApk.ServiceDispatcher>(sd);
17              }
18
19              ......
20          }
21          ......
22
23          ServiceDispatcher(ServiceConnection conn,
24                  Context context, Handler activityThread, int flags) {
```

```
25              mIServiceConnection = new InnerConnection(this);
26              mConnection = conn;
27              mContext = context;
28              mActivityThread = activityThread;
29              ......
30          }
31          ......
32
33          IServiceConnection getIServiceConnection() {
34              return mIServiceConnection;
35          }
36
37          ......
38      }
39
40      ......
41  }
```

ServiceDispatcher类有一个类型为InnerConnection的成员变量mIServiceConnection，它指向一个实现了IServiceConnection接口的Binder本地对象。

ServiceDispatcher类还有另外三个成员变量mConnection、mActivityThread和mContext，其中，成员变量mContext指向了一个Activity组件；成员变量mActivityThread和mConnection分别指向了与该Activity组件相关联的一个Handler对象和一个ServiceConnection对象。例如，Counter组件在绑定CounterService组件时，LoadedApk类就会为它创建一个ServiceDispatcher对象，这个ServiceDispatcher对象的成员变量mContext、mConnection和mActivityThread就分别指向了Counter组件、Counter组件内部的ServiceConnection对象serviceConnection和Counter组件所运行在的应用程序进程的主线程中的一个Handler对象，即ActivityThread类的成员变量mH。

最后，ServiceDispatcher类的成员函数getIServiceConnection将其内部的成员变量mIServiceConnection所指向的一个InnerConnection对象返回给调用者。

注意

这个InnerConnection对象内部有一个类型为ServiceDispatcher的弱引用mDispatcher，它指向了一个外部的ServiceDispatcher对象。

回到ContextImpl类的成员函数bindService中，它将ServiceConnection 对象conn封装成一个InnerConnection对象之后，最后就请求ActivityManagerService将CounterService组件绑定到Counter组件中。

Step 4：ActivityManagerProxy.bindService

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
01 class ActivityManagerProxy implements IActivityManager
02 {
03     ......
04
05     public int bindService(IApplicationThread caller, IBinder token,
06             Intent service, String resolvedType, IServiceConnection connection,
07             int flags) throws RemoteException {
08         Parcel data = Parcel.obtain();
09         Parcel reply = Parcel.obtain();
10         data.writeInterfaceToken(IActivityManager.descriptor);
11         data.writeStrongBinder(caller != null ? caller.asBinder() : null);
12         data.writeStrongBinder(token);
13         service.writeToParcel(data, 0);
14         data.writeString(resolvedType);
15         data.writeStrongBinder(connection.asBinder());
16         data.writeInt(flags);
```

```
17          mRemote.transact(BIND_SERVICE_TRANSACTION, data, reply, 0);
18          reply.readException();
19          int res = reply.readInt();
20          data.recycle();
21          reply.recycle();
22          return res;
23      }
24
25      ......
26  }
```

第10行到第16行代码将前面传进来的参数写入到Parcel对象data中，接着再通过ActivityManagerProxy类内部的一个Binder代理对象mRemote向ActivityManagerService发送一个类型为BIND_SERVICE_TRANSACTION的进程间通信请求。

以上4步都是在应用程序Counter中执行的，接下来第5步到第8步是在ActivityManagerService中执行的，主要是用来处理Counter组件发出的类型为BIND_SERVICE_TRANSACTION的进程间通信请求，如图8-8所示。

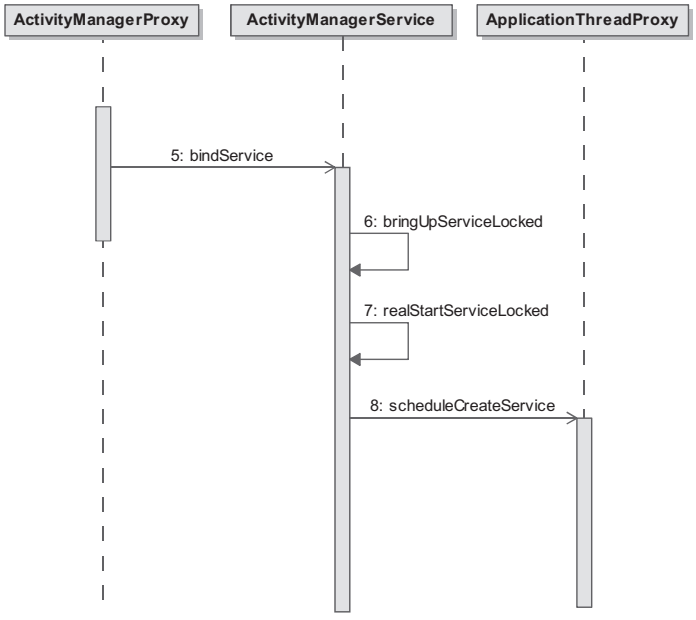


图8-8　CounterService组件的绑定过程——Step 5至Step 8

Step 5：ActivityManagerService.bindService

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02          implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03      ......
04
05      public int bindService(IApplicationThread caller, IBinder token,
06              Intent service, String resolvedType,
07              IServiceConnection connection, int flags) {
08          ......
09
10          synchronized(this) {
11              ......
12              final ProcessRecord callerApp = getRecordForAppLocked(caller);
13              ......
14
15              ActivityRecord activity = null;
16              if (token != null) {
```

```
17                  int aindex = mMainStack.indexOfTokenLocked(token);
18                  ......
19                  activity = (ActivityRecord)mMainStack.mHistory.get(aindex);
20              }
21              ......
22
23              ServiceLookupResult res =
24                  retrieveServiceLocked(service, resolvedType,
25                  Binder.getCallingPid(), Binder.getCallingUid());
26              ......
27
28              ServiceRecord s = res.record;
29              ......
30
31              AppBindRecord b = s.retrieveAppBindingLocked(service, callerApp);
32              ConnectionRecord c = new ConnectionRecord(b, activity,
33                  connection, flags, clientLabel, clientIntent);
34
35              IBinder binder = connection.asBinder();
36              ArrayList<ConnectionRecord> clist = s.connections.get(binder);
37
38              if (clist == null) {
39                  clist = new ArrayList<ConnectionRecord>();
40                  s.connections.put(binder, clist);
41              }
42              clist.add(c);
43              ......
44
45              if ((flags&Context.BIND_AUTO_CREATE) != 0) {
46                  ......
47                  if (!bringUpServiceLocked(s, service.getFlags(), false)) {
48                      return 0;
49                  }
50              }
51
52              ......
53          }
54
55          return 1;
56      }
57
58      ......
59  }
```

ActivityManagerService类的成员函数bindService用来处理类型为BIND_SERVICE_TRANSACTION的进程间通信请求。

第12行代码根据参数caller来得到一个ProcessRecord对象callerApp，用来描述正在请求ActivityManagerService执行绑定Service组件操作的一个Activity组件所运行在的应用程序进程。

第15行到第20行代码根据参数token来得到一个ActivityRecord对象activity，用来描述正在请求ActivityManagerService执行绑定Service组件操作的一个Activity组件。

第23行到第28行代码根据参数service来得到一个ServiceRecord对象s，用来描述即将被绑定的Service组件。接着第31行代码再调用ServiceRecord对象s的成员函数retrieveAppBindingLocked来得到一个AppBindRecord对象b，表示ServiceRecord对象s所描述的Service组件是绑定在ProcessRecord对象callerApp所描述的一个应用程序进程中的。

ServiceRecord类的成员函数retrieveAppBindingLocked的实现如下所示。

frameworks/base/services/java/com/android/server/am/ServiceRecord.java

```
01 class ServiceRecord extends Binder {
02     ......
03
```

```
04     final HashMap<Intent.FilterComparison, IntentBindRecord> bindings
05             = new HashMap<Intent.FilterComparison, IntentBindRecord>();
06                             // All active bindings to the service.
07     ......
08
09     public AppBindRecord retrieveAppBindingLocked(Intent intent,
10             ProcessRecord app) {
11         Intent.FilterComparison filter = new Intent.FilterComparison(intent);
12         IntentBindRecord i = bindings.get(filter);
13         if (i == null) {
14             i = new IntentBindRecord(this, filter);
15             bindings.put(filter, i);
16         }
17         AppBindRecord a = i.apps.get(app);
18         if (a != null) {
19             return a;
20         }
21         a = new AppBindRecord(this, i, app);
22         i.apps.put(app, a);
23         return a;
24     }
25
26     ......
27 }
```

当一个应用程序进程绑定了一个Service组件之后，用来描述这个应用程序进程的一个ProcessRecord对象就会被保存在用来描述这个被绑定的Service组件的一个ServiceRecord对象的成员变量bindings中。由于一个Service组件可能会被多个应用程序进程绑定，因此，用来描述这个Service组件的一个ServiceRecord对象就会使用一个IntentBindRecord对象来描述这些应用程序进程，并且以一个FilterComparison对象为关键字保存在该ServiceRecord对象的成员变量bindings中。

第11行代码首先根据参数intent来创建FilterComparison对象filter，接着第12行代码就在成员变量bindings中查找是否存在一个以FilterComparison对象filter为关键字的IntentBindRecord对象i。如果不存在，那么第14行代码就会先创建这个IntentBindRecord对象i，并且第15行代码将它保存在成员变量bindings中。

第17行代码在IntentBindRecord对象i的成员变量apps中查找是否存在一个以ProcessRecord对象app为关键字的AppBindRecord对象a。如果存在，那么就说明ProcessRecord对象app所描述的应用程序进程以前已经绑定过使用Intent对象intent来描述的一个Service组件，因此，第19行代码就直接将AppBindRecord对象a返回给调用者。如果不存在，那么第21行代码就会先创建这个AppBindRecord对象a，接着第22行代码将它保存在IntentBindRecord对象i的成员变量apps中，最后第23行代码再将AppBindRecord对象a返回给调用者。

回到ActivityManagerService类的成员函数bindService中，接着第32行和第33行代码将前面获得的AppBindRecord对象b、ActivityRecord对象activity，以及参数connection封装成一个ConnectionRecord对象c，表示ActivityRecord对象activity所描述的一个Activity组件通过参数connection绑定了ServiceRecord对象s所描述的一个Service组件，并且这个Activity组件是运行在ProcessRecord对象callerApp所描述的一个应用程序进程中的。

由于一个Service组件可能会被同一个应用程序进程中的多个Activity组件使用同一个InnerConnection对象来绑定，因此，在ActivityManagerService中，用来描述该Service组件的ServiceRecord对象就可能会对应有多个ConnectionRecord对象。在这种情况下，这些ConnectionRecord对象就会被保存在一个列表中。这个列表最终会被保存在对应的ServiceRecord对象的成员变量connection所描述的一个HashMap中，并且以它里面的ConnectionRecord对象共同使用的一个InnerConnection代理对象的IBinder接口为关键字。

从前面的调用过程可以知道，参数connection是一个InnerConnection代理对象，因此，第35行代码就可以获得它的一个IBinder接口binder。接着第36行代码就在ServiceRecord对象s的成员变量connections中检查是否存在一个以IBinder接口binder为关键字的列表clist。如果不存在，那么第39行代码就会先创建这个列表clist，接着第40行代码再将它保存在ServiceRecord对象s的成员变量connections中。接下来第42行代码就将前面所创建的ConnectionRecord对象c添加到列表clist中，表示ActivityRecord对象activity所描述的一个Activity组件通过参数connection所引用的一个InnerConnection对象绑定了ServiceRecord对象s所描述的一个Service组件。

从前面的Step 1可以知道，参数flags的Context.BIND_AUTO_CREATE位等于1，因此，最后第47行就会调用成员函数bringUpServiceLocked来启动ServiceRecord对象s所描述的一个Service组件。等到这个Service组件启动起来之后，ActivityManagerService再将它与ActivityRecord对象activity所描述的一个Activity组件绑定起来。

Step 6：ActivityManagerService.bringUpServiceLocked

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     private final boolean bringUpServiceLocked(ServiceRecord r,
06             int intentFlags, boolean whileRestarting) {
07         ......
08
09         final String appName = r.processName;
10         ProcessRecord app = getProcessRecordLocked(appName, r.appInfo.uid);
11
12         if (app != null && app.thread != null) {
13             try {
14                 realStartServiceLocked(r, app);
15                 return true;
16             } catch (RemoteException e) {
17                 ......
18             }
19         }
20
21         ......
22     }
23
24     ......
25 }
```

第9行代码首先获得ServiceRecord对象r所描述的Service组件的android:process属性，并且保存在变量appName中，接着第10行根据这个属性的值，以及ServiceRecord对象r所描述的Service组件的用户ID在ActivityManagerService中查找是否已经存在一个对应的ProcessRecord对象app。如果存在，那么就说明用来运行这个Service组件的应用程序进程已经存在了。因此，第14行就会直接调用成员函数realStartServiceLocked在ProcessRecord对象app所描述的应用程序进程中启动这个Service组件。

在我们这个场景中，即将要启动的Service组件即为CounterService组件，而启动它的Activity组件即为Counter组件。由于CounterService组件和Counter组件具有相同的android:process属性，即它们是运行在一个相同的应用程序进程中的，并且这个应用程序进程在启动Counter组件时已经创建好了，因此，接下来第14行就会直接调用ActivityManagerService类的成员函数realStartServiceLocked在ProcessRecord对象app所描述的应用程序进程中启动CounterService组件。

Step 7：ActivityManagerService.realStartServiceLocked

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     private final void realStartServiceLocked(ServiceRecord r,
06             ProcessRecord app) throws RemoteException {
07         ......
08         r.app = app;
09         ......
10
11         app.services.add(r);
12         ......
13
14         try {
15             ......
16             app.thread.scheduleCreateService(r, r.serviceInfo);
17             ......
18         } finally {
19             ......
20         }
21
22         requestServiceBindingsLocked(r);
23         ......
24     }
25
26     ......
27 }
```

第8行代码首先将ServiceRecord对象r的成员变量app设置为ProcessRecord对象app，接着第11行代码又将ServiceRecord对象r添加到ProcessRecord对象app的成员变量services中，表示ServiceRecord对象r所描述的Service组件是在ProcessRecord对象app所描述的应用程序进程中启动的。

ProcessRecord对象app的成员变量thread是一个类型为ApplicationThreadProxy的Binder代理对象，它指向了ProcessRecord对象app所描述的应用程序进程中的一个ApplicationThread对象。因此，第16行就调用它的成员函数scheduleCreateService来请求ProcessRecord对象app所描述的应用程序进程将ServiceRecord对象r所描述的Service组件启动起来。

ServiceRecord对象r所描述的Service组件启动完成之后，ActivityManagerService就需要将它连接到请求绑定它的一个Activity组件中，这是通过调用ActivityManagerService类的另外一个成员函数requestServiceBindingsLocked来实现的，如第22行代码所示。

接下来，我们先分析ServiceRecord对象r所描述的Service组件的启动过程，然后再继续分析它的连接过程。

Step 8：ApplicationThreadProxy.scheduleCreateService

第8步可以参考8.2小节中的第14步，它主要是向Counter组件所在的应用程序进程发送一个类型为SCHEDULE_CREATE_SERVICE_TRANSACTION的进程间通信请求，以便它可以将CounterService组件启动起来。

以上4步都是在ActivityManagerService中执行的，接下来第9步到第13步是在应用程序Counter中执行的，主要是用来处理ActivityManagerService发出的类型为SCHEDULE_CREATE_SERVICE_TRANSACTION的进程间通信请求，如图8-9所示。

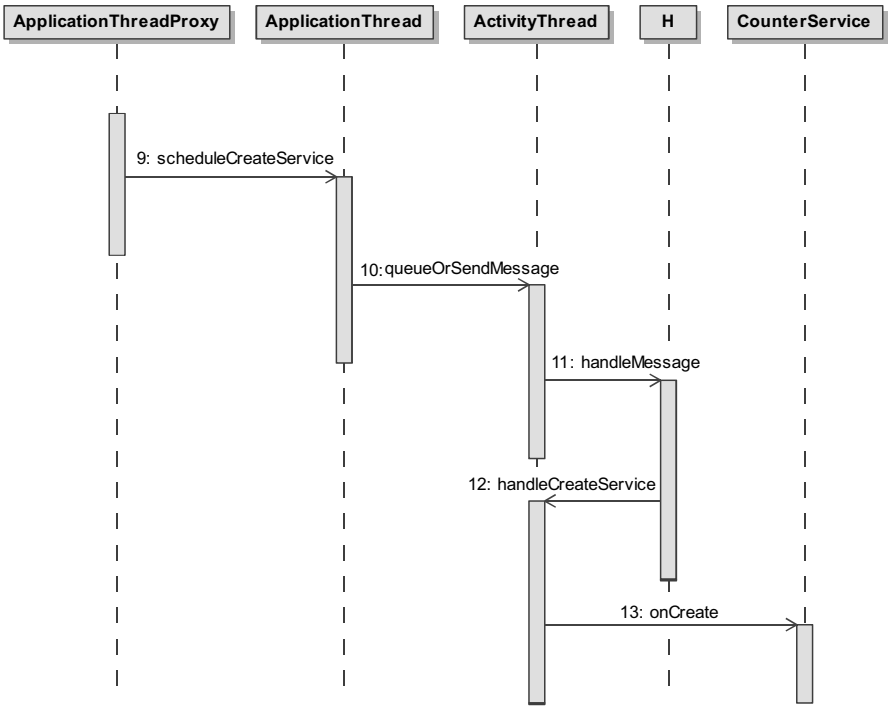


图8-9　CounterService组件的绑定过程——Step 9至Step 13

Step 9：ApplicationThread.scheduleCreateService

Step 10：ActivityThread.queueOrSendMessage

Step 11：H.handleMessage

Step 12：ActivityThread.handleCreateService

第9步到第12步可以参考8.2小节中的第15步到第18步。

Step 13：CounterService.onCreate

packages/experimental/Counter/src/shy/luo/counter/CounterService.java

```
01 public class CounterService extends Service implements ICounterService {
02     ......
03
04     @Override
05     public void onCreate() {
06         super.onCreate();
07
08         Log.i(LOG_TAG, "Counter Service Created.");
09     }
10
11     ......
12 }
```

当CounterService组件的成员函数onCreate被调用之后，CounterService组件就启动起来了。

以上5步都是在应用程序Counter中执行的，接下来第14步和第15步是在ActivityManagerService中执行的，即调用ActivityManagerService类的成员函数requestServiceBindingsLocked将CounterService组件和Counter组件连接起来，如图8-10所示。

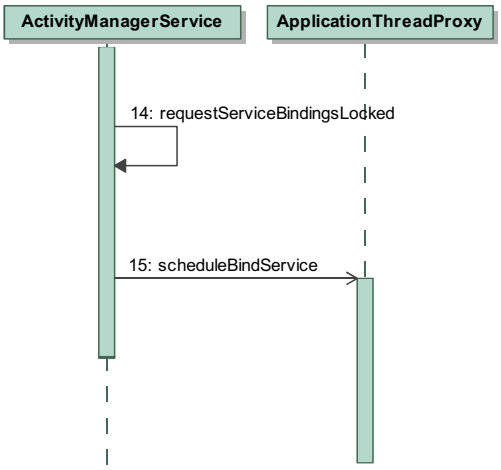


图8-10　CounterService组件的绑定过程——Step 14和Step 15

Step 14: ActivityManagerService.requestServiceBindingsLocked

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     private final void requestServiceBindingsLocked(ServiceRecord r) {
06         Iterator<IntentBindRecord> bindings = r.bindings.values().iterator();
07         while (bindings.hasNext()) {
08             IntentBindRecord i = bindings.next();
09             if (!requestServiceBindingLocked(r, i, false)) {
10                 break;
11             }
12         }
13     }
14
15     ......
16 }
```

参数r指向的一个ServiceRecord对象是在Step 5中创建的，用来描述一个已经启动起来了的Service组件，即CounterService组件。

在ServiceRecord对象r的成员变量bindings中，保存了一系列IntentBindRecord对象，每一个IntentBindRecord对象都用来描述若干个需要将ServiceRecord对象r所描述的Service组件绑定到它们里面去的应用程序进程。因此，第7行到第12行的while循环依次调用另外一个成员函数requestServiceBindingLocked将ServiceRecord对象r所描述的Service组件绑定到这些应用程序进程中。

ActivityManagerService类的成员函数requestServiceBindingLocked的实现如下所示。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     private final boolean requestServiceBindingLocked(ServiceRecord r,
06             IntentBindRecord i, boolean rebind) {
07         ......
08         if ((!i.requested || rebind) && i.apps.size() > 0) {
09           try {
10               ......
11               r.app.thread.scheduleBindService(r, i.intent.getIntent(), rebind);
12               if (!rebind) {
```

```
13                  i.requested = true;
14              }
15              ......
16          } catch (RemoteException e) {
17              ......
18          }
19      }
20      return true;
21  }
22
23  ......
24 }
```

参数rebind用来描述是否要将ServiceRecord对象r所描述的Service组件重新绑定到IntentBindRecord对象i所描述的应用程序进程中。从前面的调用过程可以知道，参数rebind的值为false，这意味着IntentBindRecord对象i所描述的应用程序进程是第一次请求绑定ServiceRecord对象r所描述的Service组件的。

ActivityManagerService在将ServiceRecord对象r所描述的Service组件绑定到IntentBindRecord对象i所描述的应用程序进程中之前，首先要获得这个Service组件内部的一个Binder本地对象，以便这些应用程序进程可以通过它来获得要绑定的Service组件的访问接口。

第8行的if语句检查ActivityManagerService是否已经为IntentBindRecord对象i所描述的应用程序进程请求过ServiceRecord对象r所描述的Service组件返回其内部的一个Binder本地对象。如果还没有请求过，即IntentBindRecord对象i的成员变量requested的值为false，并且IntentBindRecord对象i所描述的应用程序进程的个数大于0，那么第11行代码就会请求ServiceRecord对象r所描述的Service组件返回其内部的一个Binder本地对象，并且第13行代码将IntentBindRecord对象i的成员变量requested的值设置为true，避免以后重复请求。

从前面的调用过程可以知道，ServiceRecord对象r所描述的Service组件即为CounterService组件，它是运行在应用程序Counter中的，因此，接下来ActivityManagerService就会请求应用程序Counter返回CounterService组件内部的一个Binder本地对象。

Step 15：ApplicationThreadProxy.scheduleBindService

frameworks/base/core/java/android/app/ApplicationThreadNative.java

```
01 class ApplicationThreadProxy implements IApplicationThread {
02     ......
03
04     public final void scheduleBindService(IBinder token, Intent intent, boolean rebind)
05             throws RemoteException {
06         Parcel data = Parcel.obtain();
07         data.writeInterfaceToken(IApplicationThread.descriptor);
08         data.writeStrongBinder(token);
09         intent.writeToParcel(data, 0);
10         data.writeInt(rebind ? 1 : 0);
11         mRemote.transact(SCHEDULE_BIND_SERVICE_TRANSACTION, data, null,
12             IBinder.FLAG_ONEWAY);
13         data.recycle();
14     }
15
16     ......
17 }
```

第7行到第10行代码将前面传进来的参数写入到Parcel对象data中，接着再通过ApplicationThreadProxy类内部的一个Binder代理对象mRemote向应用程序Counter发送一个类型为SCHEDULE_BIND_

SERVICE_TRANSACTION的进程间通信请求。

以上两步都是在ActivityManagerService中执行的，接下来第16步到第21步是在应用程序Counter中执行的，主要是用来处理ActivityManagerService发出的类型为SCHEDULE_BIND_SERVICE_TRANSACTION的进程间通信请求，如图8-11所示。

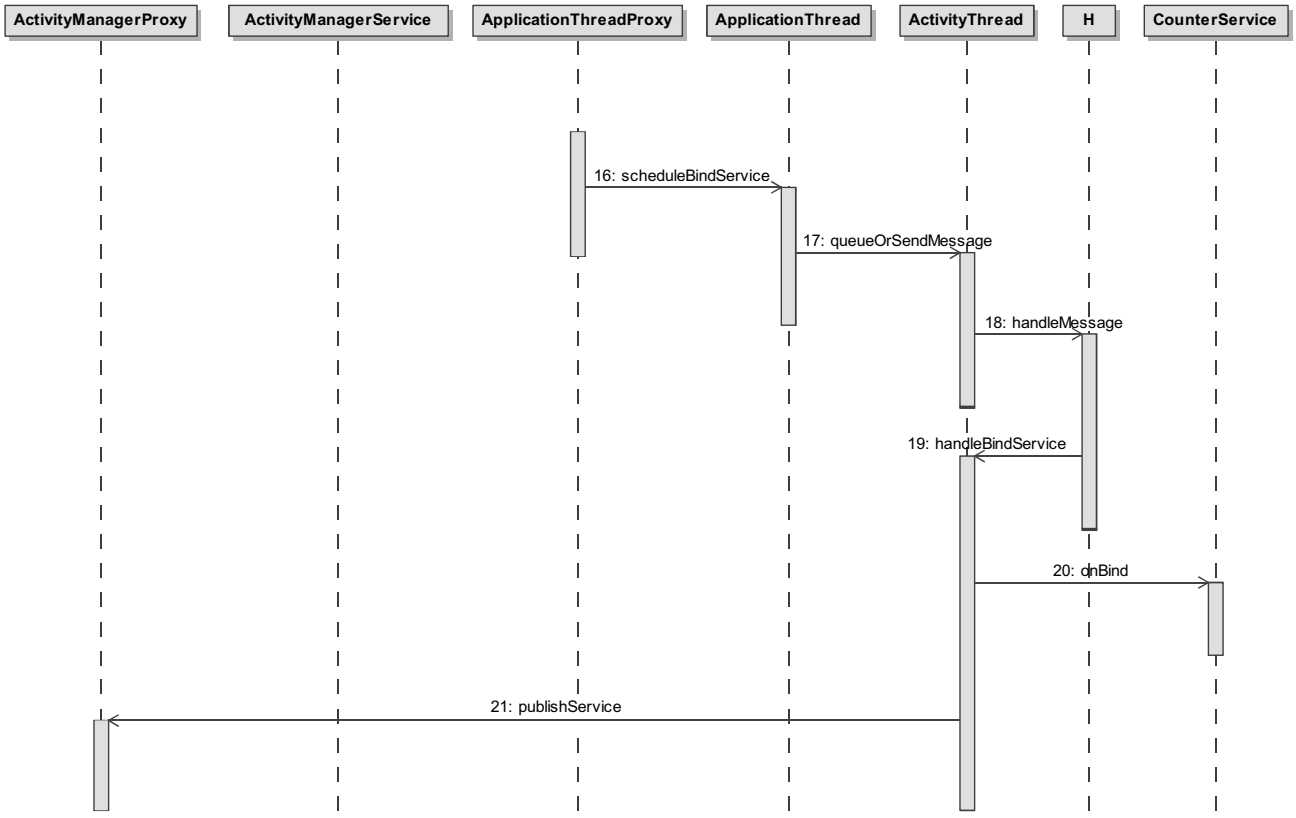


图8-11　CounterService组件的绑定过程——Step 16至Step 21

Step 16: ApplicationThread.scheduleBindService

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02      ......
03
04      private final class ApplicationThread extends ApplicationThreadNative {
05          ......
06
07          public final void scheduleBindService(IBinder token, Intent intent,
08                  boolean rebind) {
09              BindServiceData s = new BindServiceData();
10              s.token = token;
11              s.intent = intent;
12              s.rebind = rebind;
13
14              queueOrSendMessage(H.BIND_SERVICE, s);
15          }
16
17          ......
18      }
19
20      ......
21 }
```

ApplicationThread类的成员函数scheduleBindService用来处理类型为SCHEDULE_BIND_SERVICE_TRANSACTION的进程间通信请求。

第9行到第12行代码将要返回内部的一个Binder本地对象的一个Service组件的信息封装成的一个BindServiceData对象，接着再调用ActivityThread类的成员函数queueOrSendMessage将这个BindServiceData对象封装成一个类型为BIND_SERVICE的消息，以便可以将它发送到当前应用程序进程的主线程的消息队列中。

Step 17：ActivityThread.queueOrSendMessage

第17步可以参考7.2小节中的第31步，它主要是将上一步创建的BindServiceData对象封装成一个类型为BIND_SERVICE的消息，然后调用ActivityThread类的成员变量mH的成员函数sendMessage将这个消息发送到应用程序Counter的主线程的消息队列中。这个消息最终是在ActivityThread类的成员变量mH的成员函数handleMessage中处理的。

Step 18：H.handleMessage

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final class H extends Handler {
05         ......
06
07         public void handleMessage(Message msg) {
08             ......
09             switch (msg.what) {
10             ......
11             case BIND_SERVICE:
12                 handleBindService((BindServiceData)msg.obj);
13                 break;
14             ......
15             }
16         }
17     }
18
19     ......
20 }
```

第12行代码首先将Message对象msg的成员变量obj转换成一个BindServiceData对象，它里面包含了要获得其内部的一个Binder本地对象的一个Service组件的信息，接着再调用ActivityThread类的成员函数handleBindService来获得这个Service组件内部的一个Binder本地对象。

Step 19：ActivityThread.handleBindService

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final void handleBindService(BindServiceData data) {
05         Service s = mServices.get(data.token);
06         if (s != null) {
07             try {
08                 ......
09                 try {
10                     if (!data.rebind) {
11                         IBinder binder = s.onBind(data.intent);
12                         ActivityManagerNative.getDefault().publishService(
13                             data.token, data.intent, binder);
14                     } else {
15                         ......
```

```
16                      }
17                      ......
18                  } catch (RemoteException ex) {
19                  }
20              } catch (Exception e) {
21                  ......
22              }
23          }
24      }
25
26      ......
27  }
```

BindServiceData对象data的成员变量token指向了一个Binder代理对象，它引用了ActivityManagerService中的一个ServiceRecord对象，而这个ServiceRecord对象是用来描述应用程序Counter中的CounterService组件的。

在前面的Step 12中，CounterService组件在应用程序Counter中启动起来之后，就会将自己保存在ActivityThread类的成员变量mServices中，并且是以BindServiceData对象data的成员变量token所指向的一个Binder代理对象为关键字的。因此，第5行就可以通过BindServiceData对象data的成员变量token来获得一个用来描述CounterService组件的Service对象。

从前面的调用过程可以知道，BindServiceData对象data的成员变量rebind的值为false，因此，接下来第11行就会先调用CounterService组件的成员函数onBind来获得其内部的一个Binder本地对象，接着第12行和第13行再调用ActivityManagerService代理对象的成员函数publishService将这个Binder本地对象传递给ActivityManagerService。

接下来，我们就先分析CounterService组件的成员函数onBind的实现，然后再继续分析ActivityManagerService代理对象的成员函数publishService的实现。

Step 20：CounterService.onBind

packages/experimental/Counter/src/shy/luo/counter/CounterService.java

```
01 public class CounterService extends Service implements ICounterService {
02      ......
03
04      private final IBinder binder = new CounterBinder();
05
06      public class CounterBinder extends Binder {
07          public CounterService getService() {
08              return CounterService.this;
09          }
10      }
11
12      @Override
13      public IBinder onBind(Intent intent) {
14          return binder;
15      }
16
17      ......
18  }
```

CounterService组件的成员函数onBind将内部一个类型为CounterBinder的Binder本地对象返回给调用者。CounterBinder类有一个成员函数getService，通过它可以获得CounterService组件的访问接口。

回到前面的Step 19中，即ActivityThread类的成员函数handleBindService中，接下来就会调用ActivityManagerService代理对象的成员函数publishService将前面获得的一个Binder本地对象传递给ActivityManagerService。

Step 21：ActivityManagerProxy.publishService

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
01 class ActivityManagerProxy implements IActivityManager
02 {
03     ......
04
05     public void publishService(IBinder token,
06     Intent intent, IBinder service) throws RemoteException {
07         Parcel data = Parcel.obtain();
08         Parcel reply = Parcel.obtain();
09         data.writeInterfaceToken(IActivityManager.descriptor);
10         data.writeStrongBinder(token);
11         intent.writeToParcel(data, 0);
12         data.writeStrongBinder(service);
13         mRemote.transact(PUBLISH_SERVICE_TRANSACTION, data, reply, 0);
14         reply.readException();
15         data.recycle();
16         reply.recycle();
17     }
18
19     ......
20 }
```

第9行到第12行代码将前面传进来的参数写入到Parcel对象data中，接着再通过ActivityManagerProxy类内部的一个Binder代理对象mRemote向ActivityManagerService发送一个类型为PUBLISH_SERVICE_TRANSACTION的进程间通信请求。

以上6步都是在应用程序Counter中执行的，接下来第22步是在ActivityManagerService中执行的，主要是用来处理CounterService组件发出的类型为PUBLISH_SERVICE_TRANSACTION的进程间通信请求，如图8-12所示。

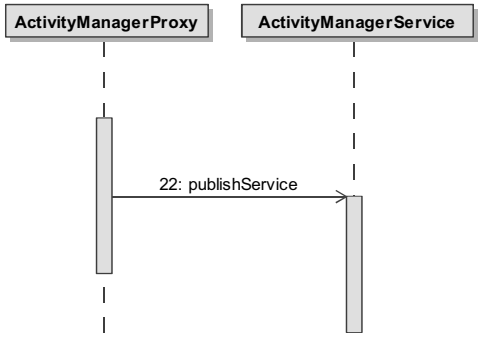


图8-12　CounterService组件的绑定过程——Step 22

Step 22：ActivityManagerService.publishService

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     public void publishService(IBinder token, Intent intent, IBinder service) {
06         ......
07         synchronized(this) {
08             ......
09             ServiceRecord r = (ServiceRecord)token;
10             ......
11
12             if (r != null) {
```

```
13                  Intent.FilterComparison filter
14                      = new Intent.FilterComparison(intent);
15                  IntentBindRecord b = r.bindings.get(filter);
16                  if (b != null && !b.received) {
17                      b.binder = service;
18                      b.requested = true;
19                      b.received = true;
20                      if (r.connections.size() > 0) {
21                          Iterator<ArrayList<ConnectionRecord>> it
22                              = r.connections.values().iterator();
23                          while (it.hasNext()) {
24                              ArrayList<ConnectionRecord> clist = it.next();
25                              for (int i=0; i<clist.size(); i++) {
26                                  ConnectionRecord c = clist.get(i);
27                                  ......
28                                  try {
29                                      c.conn.connected(r.name, service);
30                                  } catch (Exception e) {
31                                      ......
32                                  }
33                              }
34                          }
35                      }
36                  }
37
38                  ......
39              }
40          }
41      }
42
43      ......
44  }
```

ActivityManagerService类的成员函数publishService用来处理类型为PUBLISH_SERVICE_TRANSACTION的进程间通信请求。

参数token指向的是一个ServiceRecord对象，用来描述Counter组件请求绑定的CounterService组件；参数intent指向的是一个Intent对象，Counter组件最初就是通过它来请求ActivityManagerService绑定CounterService组件的；参数service指向的是CounterService组件内部的一个Binder本地对象。

第9行代码首先根据参数token获得一个ServiceRecord对象r，接着第13行到第15行代码再根据参数intent在ServiceRecord对象r的成员变量bindings中获得一个IntentBindRecord对象b。ServiceRecord对象r和IntentBindRecord对象b都是在前面的Step 5中创建的。

IntentBindRecord对象b的成员变量received用来描述ServiceRecord对象r所描述的Service组件是否已经将其内部的一个Binder本地对象传递给ActivityManagerService了。如果还没有传递，那么它的值就等于false，接下来第17行代码就会将这个Binder本地对象，即参数service指向的一个Binder本地对象保存在IntentBindRecord对象b的成员变量binder中，并且第18行和第19行代码会将IntentBindRecord对象b的成员变量requested和received的值都设置为true，避免ActivityManagerService以后重复请求ServiceRecord对象r所描述的Service组件返回其内部的一个Binder本地对象。

每一个需要绑定ServiceRecord对象r所描述的Service组件的Activity组件都使用一个ConnectionRecord对象来描述。由于不同的Activity组件可能会使用相同的InnerConnection对象来绑定ServiceRecord对象r所描述的Service组件，因此，ActivityManagerService就会把这些使用了同一个InnerConnection对象的ConnectionRecord对象放在同一个列表中。这样ActivityManagerService就会得到与ServiceRecord对象r相关的一系列ConnectionRecord对象列表，它们最终保存在ServiceRecord对象r的成员变量connections所描述的一个HashMap中，并且以它们所使用的InnerConnection对象为关键字。

第23行到第34行的while循环依次遍历保存在ServiceRecord对象r的成员变量connections中的每一个ConnectionRecord对象，并且第29行会调用它们的成员变量conn的成员函数connected将参数service所描述的一个Binder本地对象传递给与它们关联的Activity组件，以便这些Activity组件可以通过这个Binder本地对象来获得它们所要绑定的Service组件的访问接口。

ConnectionRecord类的成员变量conn是一个类型为IServiceConnection的Binder代理对象，它引用了一个类型为InnerConnection的Binder本地对象。这个Binder本地对象是用来连接一个Service组件和一个Activity组件的，并且与这个Activity组件运行在同一个应用程序进程中。在我们这个场景中，这个Activity组件即为应用程序Counter中的Counter组件，而它要绑定的Service组件即为同一个应用程序中的CounterService组件。因此，第29行代码实际上就是调用Counter组件所使用的一个InnerConnection对象的成员函数connected来连接CounterService组件，以便可以获得CounterService组件内部的一个Binder本地对象。

以上一步是在ActivityManagerService中执行的，接下来第23步到第27步是在应用程序Counter中执行的，主要是用来连接Counter组件和CounterService组件，如图8-13所示。

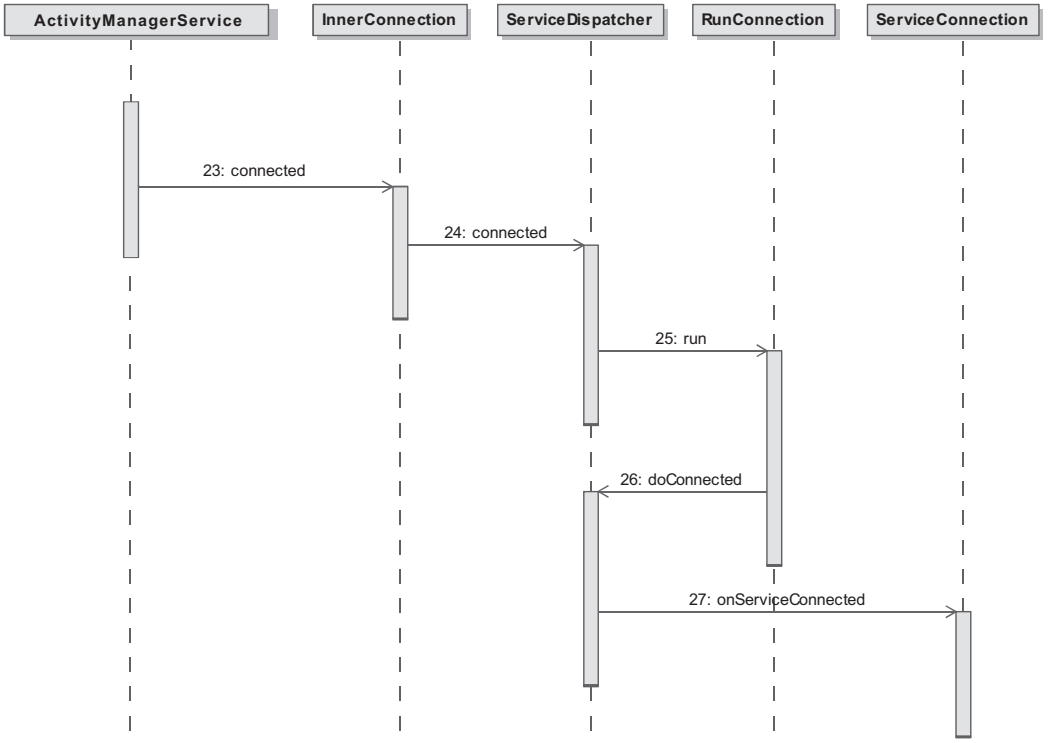


图8-13　CounterService组件的绑定过程——Step 23至Step 27

Step 23：InnerConnection.connected

frameworks/base/core/java/android/app/LoadedApk.java

```
01 final class LoadedApk {
02     ......
03
04     static final class ServiceDispatcher {
05         ......
06
07         private static class InnerConnection extends IServiceConnection.Stub {
08             final WeakReference<LoadedApk.ServiceDispatcher> mDispatcher;
09             ......
10
11             public void connected(ComponentName name, IBinder service) throws RemoteException {
```

```
12                  LoadedApk.ServiceDispatcher sd = mDispatcher.get();
13                  if (sd != null) {
14                      sd.connected(name, service);
15                  }
16              }
17              ......
18          }
19
20          ......
21      }
22
23      ......
24  }
```

在前面的Step 3中提到，InnerConnection类的成员变量mDispatcher是一个类型为ServiceDispatcher的弱引用，因此，第14行代码实际上是调用了ServiceDispatcher类的成员函数connected来连接Counter组件和CounterService组件。

Step 24：ServiceDispatcher.connected

frameworks/base/core/java/android/app/LoadedApk.java

```
01 final class LoadedApk {
02      ......
03
04      static final class ServiceDispatcher {
05          ......
06          private final Handler mActivityThread;
07          ......
08
09          public void connected(ComponentName name, IBinder service) {
10              if (mActivityThread != null) {
11                  mActivityThread.post(new RunConnection(name, service, 0));
12              }
13              ......
14          }
15
16          ......
17      }
18
19      ......
20  }
```

在前面的Step 3中提到，ServiceDispatcher类的成员变量mActivityThread指向了ActivityThread类的成员变量mH，它是用来向应用程序Counter的主线程的消息队列发送消息的。

第11行代码首先将参数name和service封装成一个RunConnection对象，然后再将这个RunConnection对象封装成一个消息，最后将这个消息发送到应用程序Counter的主线程的消息队列中。这个消息最终是在RunConnection类的成员函数run中处理的。

Step 25：RunConnection.run

frameworks/base/core/java/android/app/LoadedApk.java

```
01 final class LoadedApk {
02      ......
03
04      static final class ServiceDispatcher {
05          ......
06
07         private final class RunConnection implements Runnable {
08              RunConnection(ComponentName name, IBinder service, int command) {
09                  mName = name
10                  mService = service;
```

```
11                  mCommand = command;
12              }
13
14              public void run() {
15                  if (mCommand == 0) {
16                      doConnected(mName, mService);
17                  }
18                  ......
19              }
20
21              final ComponentName mName;
22              final IBinder mService;
23              final int mCommand;
24          }
25
26          ......
27      }
28
29      ......
30  }
```

第16行调用了ServiceDispatcher类的成员函数doConnected将RunConnection类内部的成员变量mService所描述的一个Binder本地对象传递给Counter组件。从前面的调用过程可以知道，这个Binder本地对象即为CounterService组件内部的Binder本地对象。

注意

第24步为什么不直接调用ServiceDispatcher类的成员函数doConnected将CounterService组件内部的Binder本地对象传递给Counter组件，而是要通过消息处理机制来间接传递呢？一是因为当前线程需要尽快返回到Binder线程池中，以便可以处理其他的Binder进程间通信请求，从而提高应用程序Counter并发处理Binder进程间通信请求的能力；二是因为Counter组件在接收CounterService组件内部的Binder本地对象时，可能会需要执行与用户界面相关的操作，因此就需要通过消息处理机制将这个传递操作放在主线程中执行。

Step 26：ServiceDispatcher.doConnected

frameworks/base/core/java/android/app/LoadedApk.java

```
01 final class LoadedApk {
02      ......
03
04      static final class ServiceDispatcher {
05          ......
06          private final ServiceConnection mConnection;
07          ......
08
09          public void doConnected(ComponentName name, IBinder service) {
10              ......
11
12              // If there is a new service, it is now connected.
13              if (service != null) {
14                  mConnection.onServiceConnected(name, service);
15              }
16          }
17
18          ......
19      }
20
21      ......
22  }
```

从前面的Step 3可以知道，在我们这个场景中，ServiceDispatcher类的成员变量mConnection指向的是Counter组件内部的成员变量serviceConnection所描述的一个自定义ServiceConnection对象。因此，第14行实际上是调用了Counter组件内部的成员变量serviceConnection的成员函数onServiceConnected，以便可以将参数service所描述的一个Binder本地对象传递给Counter组件。

Step 27：ServiceConnection.onServiceConnected

packages/experimental/Counter/src/shy/luo/counter/Counter.java

```
01 public class Counter extends Activity implements OnClickListener, ICounterCallback {
02     ......
03
04     private ICounterService counterService = null;
05     ......
06
07     private ServiceConnection serviceConnection = new ServiceConnection() {
08         public void onServiceConnected(ComponentName className, IBinder service) {
09             counterService = ((CounterService.CounterBinder)service).getService();
10
11             Log.i(LOG_TAG, "Counter Service Connected");
12         }
13         ......
14     };
15 }
```

参数service所描述的Binder本地对象即为CounterService组件内部的一个类型为CounterBinder的Binder本地对象。由于Counter组件和CounterService组件是运行在同一个应用程序进程中的，因此，第9行代码就可以先将参数service转换成一个CounterBinder对象，然后再调用它的成员函数getService来获得CounterService组件的访问接口，最后将这个访问接口保存在Counter组件的成员变量counterService中。

至此，Counter组件绑定CounterService组件的过程就分析完成了。Counter组件获得了CounterService组件的访问接口之后，以后就可以直接使用CounterService组件提供的计数器服务了。

第 9 章

Android系统广播机制

在Android系统中，广播（Broadcast）是一种在组件之间进行消息传递的方式。这些组件可以运行在同一个进程中，也可以运行在不同的进程中。当两个不在同一个进程中的组件通过广播机制来传递消息时，广播机制就有点类似Binder进程间通信机制。事实上，广播机制就是在Binder进程间通信机制的基础上实现的。

既然如此，Android系统为什么还需要提供广播机制呢？我们知道，在Binder进程间通信机制中，Client组件在和Service组件通信之前，必须要先获得它的一个代理对象，即Client组件事先要知道Service组件的存在。然而，在广播机制中，广播发送者事先是不需要知道广播接收者的存在的，这样就可以降低广播发送者和广播接收者之间的耦合度，进而提高系统的可扩展性和可维护性。因此，Android系统需要提供广播机制。

广播机制是一种基于消息发布和订阅的事件驱动模型，即广播发送者负责发布消息，而广播接收者需要先订阅消息，然后才能接收到消息。Android系统将广播接收者抽象为一种Broadcast Receiver组件，它是Android应用程序的四大组件之一。同时， Android应用程序的另外两种组件，即Activity组件和Service组件被赋予了发送广播的能力。因此，在Android系统中，使用广播机制在组件之间传递消息是非常方便的。

广播机制存在一个注册中心，它是由ActivityManagerService来担当的。广播接收者订阅消息的表现形式就是将自己注册到ActivityManagerService中，并且指定要接收的广播的类型。当广播发送者向广播接收者发送一个广播时，这个广播首先发送到ActivityManagerService，然后ActivityManagerService根据这个广播的类型找到相应的广播接收者，最后将这个广播发送给它们处理。

广播接收者的注册方式分为静态注册和动态注册两种。在静态注册方式中，用来描述广播接收者的Broadcast Receiver组件必须要在配置文件AndroidManifest.xml中注明它们所感兴趣的广播的类型，以便ActivityManagerService可以找到它们。在动态注册方式中，我们需要在代码中手动地调用Context接口的成员函数registerReceiver将Broadcast Receiver组件注册到ActivityManagerService中。Activity组件和Service组件都实现了Context接口，因此，我们可以方便地在一个Activity组件或者Service组件中注册一个Broadcast Receiver组件。在同等情况下，动态注册的广播接收者要比静态注册的广播接收者优先接收到广播。

广播的发送方式分为有序和无序两种。我们在注册广播接收者时，可以指定它们的优先级。当ActivityManagerService接收到有序广播时，它就会先将这个有序广播发送给符合条件的、优先级较高的广播接收者处理，然后再发送给符合条件的、优先级较低的广播接收者处理；而当ActivityManagerService接收到无序广播时，它就会忽略广播接收者的优先级，并行地将这个无序广播发送给所有符合条件的广播接收者处理。

接下来，我们就首先开发一个广播机制应用实例，然后结合这个应用实例来分析Android系统的广播机制，包括分析广播接收者的注册过程，以及广播的发送过程。

9.1 广播机制应用实例

在前面的8.1小节中，我们开发了一个应用程序Counter，它由一个Activity组件Counter和一个Service组件CounterService组成。Counter组件在启动CounterService组件的计数器时，需要指定一个计数器回调接口ICounterCallback，以便CounterService组件可以将计数器的当前计数实时地更新到Counter组件的用户界面上。这种数据更新方式存在一定的缺陷，因为这会使得CounterService组件依赖于Counter组件提供的计数器回调接口ICounterCallback。

本节将开发一个名称为Broadcounter的应用程序，它实现的功能与应用程序Counter是一样的，不过它使用广播机制来代替计数器回调接口ICounterCallbak，使得CounterService组件不必依赖于Counter组件。

应用程序Broadcounter的目录结构如下：

```
~/Android/packages/experimental/Broadcounter
----AndroidManifest.xml
----Android.mk
----src
    ----shy/luo/broadcounter
        ----ICounterService.java
        ----CounterService.java
        ----Broadcounter.java
----res
    ----layout
        ----main.xml
    ----values
        ----strings.xml
    ----drawable
          ----icon.png
```

它包含了一个源代码目录src、一个资源目录res、一个配置文件AndroidManifest.xml和一个编译脚本文件Android.mk。接下来我们分别介绍每一个文件的内容。

ICounterService.java

```
1 package shy.luo.broadcounter;
2
3 public interface ICounterService {
4     public void startCounter(int initVal);
5     public void stopCounter();
6 }
```

与前面8.1小节中的ICounterService.java文件类似，这个文件定义了一个计数器接口ICounterService，用来启动和停止计数器，不过，在启动计数器时，我们只需要指定计数器的初始值即可，而不必指定计数器回调接口。

CounterService.java

```
1 package shy.luo.broadcounter;
2
3 import android.app.Service;
4 import android.content.Intent;
5 import android.os.AsyncTask;
```

```
6 import android.os.Binder;
7 import android.os.IBinder;
8 import android.util.Log;
9
10 public class CounterService extends Service implements ICounterService {
11     private final static String LOG_TAG = "shy.luo.broadcounter.CounterService";
12
13     public final static String BROADCAST_COUNTER_ACTION = "shy.luo.broadcounter.COUNTER_ACTION";
14     public final static String COUNTER_VALUE = "shy.luo.broadcounter.counter.value";
15
16     private boolean stop = false;
17     private final IBinder binder = new CounterBinder();
18
19     public class CounterBinder extends Binder {
20         public CounterService getService() {
21             return CounterService.this;
22         }
23     }
24
25     @Override
26     public IBinder onBind(Intent intent) {
27         return binder;
28     }
29
30     @Override
31     public void onCreate() {
32         super.onCreate();
33         Log.i(LOG_TAG, "Counter Service Created.");
34     }
35
36     public void startCounter(int initVal) {
37         AsyncTask<Integer, Integer, Integer> task = new AsyncTask<Integer, Integer, Integer>() {
38             @Override
39             protected Integer doInBackground(Integer... vals) {
40                 Integer initCounter = vals[0];
41
42                 stop = false;
43                 while(!stop) {
44                     publishProgress(initCounter);
45
46                     try {
47                         Thread.sleep(1000);
48                     } catch (InterruptedException e) {
49                         e.printStackTrace();
50                     }
51
52                     initCounter++;
53                 }
54
55                 return initCounter;
56             }
57
58             @Override
59             protected void onProgressUpdate(Integer... values) {
60                 super.onProgressUpdate(values);
61
62                 int counter = values[0];
63                 Intent intent = new Intent(BROADCAST_COUNTER_ACTION);
64                 intent.putExtra(COUNTER_VALUE, counter);
65
66                 sendBroadcast(intent);
67             }
68
```

```
69              @Override
70              protected void onPostExecute(Integer val) {
71                  int counter = val;
72                  Intent intent = new Intent(BROADCAST_COUNTER_ACTION);
73                  intent.putExtra(COUNTER_VALUE, counter);
74
75                  sendBroadcast(intent);
76              }
77
78          };
79
80          task.execute(initVal);
81      }
82
83      public void stopCounter() {
84          stop = true;
85      }
86  }
```

与前面8.1小节中的CounterService.java文件类似，这个文件实现了一个CounterService组件，它也是通过异步任务（AsyncTask）来提供计数器功能的，唯一的区别在于这个文件实现的CounterService组件是通过广播的形式将计数器的当前计数值实时地更新到Counter组件的用户界面上的。

在计数器的执行过程中，即在异步任务task的成员函数onProgressUpdate中，CounterService组件会先将计数器的当前计数值写入到一个类型为BROADCAST_COUNTER_ACTION的广播中，接着再调用其父类ContextWrapper的成员函数sendBroadcast将这个广播发送出去，以便Counter组件可以接收到这个广播，并且获得里面的计数值。

同样，在计数器停止执行时，即在异步任务task的成员函数onPostExecute中，CounterService组件也会先将计数器的最后计数值写入到一个类型为BROADCAST_COUNTER_ACTION的广播中，接着再调用其父类ContextWrapper的成员函数sendBroadcast将这个广播发送出去，以便Counter组件可以接收到这个广播，并且获得里面的计数值。

Broadcounter.java

```
01 package shy.luo.broadcounter;
02
03 import android.app.Activity;
04 import android.content.BroadcastReceiver;
05 import android.content.ComponentName;
06 import android.content.Context;
07 import android.content.Intent;
08 import android.content.IntentFilter;
09 import android.content.ServiceConnection;
10 import android.os.Bundle;
11 import android.os.IBinder;
12 import android.util.Log;
13 import android.view.View;
14 import android.view.View.OnClickListener;
15 import android.widget.Button;
16 import android.widget.TextView;
17
18 public class Broadcounter extends Activity implements OnClickListener {
19     private final static String LOG_TAG = "shy.luo.broadcounter.Broadcounter";
20
21     private Button startButton = null;
22     private Button stopButton = null;
23     private TextView counterText = null;
24
25     private ICounterService counterService = null;
```

```
26
27      @Override
28      public void onCreate(Bundle savedInstanceState) {
29          super.onCreate(savedInstanceState);
30          setContentView(R.layout.main);
31
32          startButton = (Button)findViewById(R.id.button_start);
33          stopButton = (Button)findViewById(R.id.button_stop);
34          counterText = (TextView)findViewById(R.id.textview_counter);
35
36          startButton.setOnClickListener(this);
37          stopButton.setOnClickListener(this);
38
39          Intent bindIntent = new Intent(Broadcounter.this, CounterService.class);
40          bindService(bindIntent, serviceConnection, Context.BIND_AUTO_CREATE);
41
42          Log.i(LOG_TAG, "Broadcounter Activity Created.");
43      }
44
45      @Override
46      public void onResume() {
47          super.onResume();
48          IntentFilter counterActionFilter = new IntentFilter(CounterService.BROADCAST_COUNTER_ACTION);
49          registerReceiver(counterActionReceiver, counterActionFilter);
50      }
51
52      @Override
53      public void onPause() {
54          super.onPause();
55          unregisterReceiver(counterActionReceiver);
56      }
57
58      @Override
59      public void onDestroy() {
60          super.onDestroy();
61          unbindService(serviceConnection);
62      }
63
64      @Override
65      public void onClick(View v) {
66          if(v.equals(startButton)) {
67              if(counterService != null) {
68                  counterService.startCounter(0);
69                  startButton.setEnabled(false);
70                  stopButton.setEnabled(true);
71              }
72          } else if(v.equals(stopButton)) {
73              if(counterService != null) {
74                  counterService.stopCounter();
75                  startButton.setEnabled(true);
76                  stopButton.setEnabled(false);
77              }
78          }
79      }
80
81      private ServiceConnection serviceConnection = new ServiceConnection() {
82          public void onServiceConnected(ComponentName className, IBinder service) {
83              counterService = ((CounterService.CounterBinder)service).getService();
84              Log.i(LOG_TAG, "Counter Service Connected");
85          }
86          public void onServiceDisconnected(ComponentName className) {
```

```
87              counterService = null;
88              Log.i(LOG_TAG, "Counter Service Disconnected");
89          }
90      };
91
92      private BroadcastReceiver counterActionReceiver = new BroadcastReceiver(){
93          public void onReceive(Context context, Intent intent) {
94              int counter = intent.getIntExtra(CounterService.COUNTER_VALUE, 0);
95              String text = String.valueOf(counter);
96              counterText.setText(text);
97          }
98      };
99  }
```

与前面8.1小节中的Counter.java文件类似，这个文件实现了一个Broadcounter组件，它在启动时，会调用其父类ContextWrapper的成员函数bindService来绑定CounterService组件，以便可以获得它的一个访问接口。唯一的区别在于这个文件实现的Broadcounter组件是通过一个广播接收者来获得CounterService组件内部的计数器的当前计数值的。

Broadcounter组件在内部定义了一个广播接收者counterActionReceiver，它是用来接收类型为BROADCAST_COUNTER_ACTION的广播的，以便可以获得CounterService组件的计数器的当前计数值，并且将它显示在用户界面上。

当Broadcounter组件被激活时，即在它的成员函数onResume中，它会调用父类ContextWrapper的成员函数registerReceiver将广播接收者counterActionReceiver注册到ActivityManageService中，并且指定要接收的广播的类型为BROADCAST_COUNTER_ACTION。

当Broadcounter组件被中止时，即在它的成员函数onPause中，它会调用父类ContextWrapper的成员函数unregisterReceiver来注销前面所注册的广播接收者counterActionReceiver，这样它就不会再接收到类型为BROADCAST_COUNTER_ACTION的广播了。

main.xml

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
03     android:orientation="vertical"
04     android:layout_width="fill_parent"
05     android:layout_height="fill_parent"
06     android:gravity="center">
07     <LinearLayout android:layout_width="fill_parent"
08         android:layout_height="wrap_content"
09         android:layout_marginBottom="10px"
10         android:orientation="horizontal"
11         android:gravity="center">
12             <TextView android:layout_width="wrap_content"
13                 android:layout_height="wrap_content"
14                 android:layout_marginRight="4px"
15                 android:gravity="center"
16                 android:text="@string/counter">
17             </TextView>
18             <TextView android:id="@+id/textview_counter"
19                 android:layout_width="wrap_content"
20                 android:layout_height="wrap_content"
21                 android:gravity="center"
22                 android:text="@string/init_counter">
23             </TextView>
24     </LinearLayout>
```

```
25     <LinearLayout android:layout_width="fill_parent"
26         android:layout_height="wrap_content"
27         android:orientation="horizontal"
28         android:gravity="center">
29             <Button android:id="@+id/button_start"
30                 android:layout_width="wrap_content"
31                 android:layout_height="wrap_content"
32                 android:gravity="center"
33                 android:text="@string/start">
34             </Button>
35             <Button android:id="@+id/button_stop"
36                 android:layout_width="wrap_content"
37                 android:layout_height="wrap_content"
38                 android:gravity="center"
39                 android:text="@string/stop" >
40             </Button>
41         </LinearLayout>
42 </LinearLayout>
```

这是Broadcounter组件的界面配置文件，它在屏幕中间显示两个按钮和一个文本框。

strings.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3     <string name="app_name">Broadcounter</string>
4     <string name="counter">Counter: </string>
5     <string name="init_counter">0</string>
6     <string name="start">Start Counter</string>
7     <string name="stop">Stop Counter</string>
8 </resources>
```

这是应用程序Broadcounter的字符串资源文件，定义了在应用程序中用到的各个字符串。

icon.png

这是应用程序Broadcounter的图标，可以根据需要来放置不同的图片文件。

AndroidManifest.xml

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
03     package="shy.luo.broadcounter"
04     android:versionCode="1"
05     android:versionName="1.0">
06     <application android:icon="@drawable/icon" android:label="@string/app_name">
07          <activity android:name=".Broadcounter"
08              android:label="@string/app_name">
09              <intent-filter>
10                  <action android:name="android.intent.action.MAIN" />
11                  <category android:name="android.intent.category.LAUNCHER" />
12              </intent-filter>
13          </activity>
14          <service android:name=".CounterService"
15              android:enabled="true">
16          </service>
17     </application>
18 </manifest>
```

这是应用程序Broadcounter的配置文件。由于在程序中使用到了Activity组件Broadcounter和Service组件CounterService，因此，我们就需要在这里对它们进行配置。

Android.mk

```
1 LOCAL_PATH:= $(call my-dir)
2 include $(CLEAR_VARS)
3 LOCAL_MODULE_TAGS := optional
4 LOCAL_SRC_FILES := $(call all-subdir-java-files)
5 LOCAL_PACKAGE_NAME := Broadcounter
6 include $(BUILD_PACKAGE)
```

这是应用程序Broadcounter的编译脚本文件。

上述文件都准备好之后，我们就可以对应用程序Broadcounter进行编译和打包了。

```
USER@MACHINE:~/Android$ mmm ./packages/experimental/Broadcounter/
USER@MACHINE:~/Android$ make snod
```

编译成功之后，就可以在out/target/product/gerneric/system/app目录下看到应用程序文件Broadcounter.apk了；打包成功之后，应用程序Broadcounter就包含在out/target/product/gerneric目录下的Android系统镜像文件system.img中了。

最后，我们使用新得到的系统镜像文件system.img来启动Android模拟器。

```
USER@MACHINE:~/Android$ emulator
```

Android模拟器运行起来之后，我们就可以在应用程序启动器中启动应用程序Broadcounter了，即将它的根Activity组件Broadcounter启动起来，它的界面如图9-1所示。

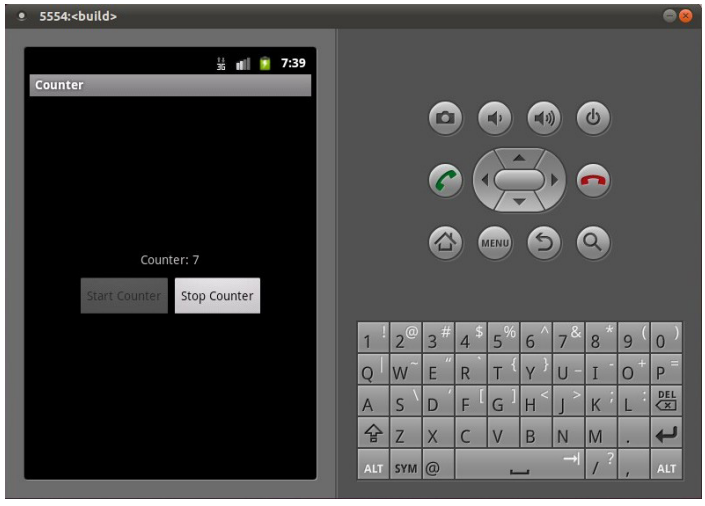


图9-1　应用程序Broadcounter的界面

点击用户界面上的Start Counter按钮，就可以将CounterService组件中的计数器启动起来，并且每隔1秒就可以看到这个计数器的计数增加1；而点击用户界面上的Stop Counter按钮，就可以停止CounterService组件中的计数器。

9.2　广播接收者的注册过程

前面提到，在Android系统的广播机制中，ActivityManagerService扮演着注册中心的角色，用来注册广播接收者以及转发广播。因此，Android应用程序注册广播接收者的过程实际上就是把一个广播接收者注册到ActivityManagerService中的过程。

在前面的9.1小节中，Activity组件Broadcounter调用其父类ContextWrapper的成员函数registerReceiver

向ActivityManagerService注册了一个广播接收者counterActionReceiver，用来接收类型为BROADCAST_COUNTER_ACTION的广播。本节将结合这个场景来分析广播接收者的注册过程，如图9-2所示。

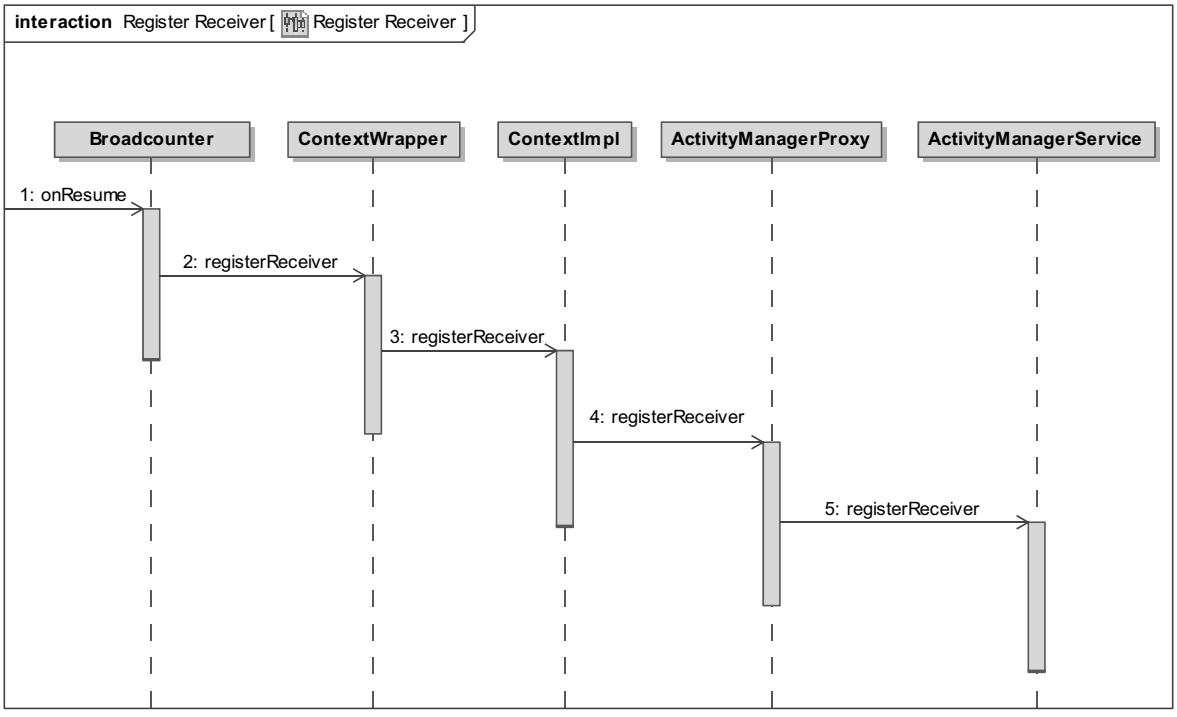


图9-2 广播接收者的注册过程

这个过程一共分为5个步骤，接下来我们就详细分析每一个步骤。

Step 1：Broadcounter.onResume

packages/experimental/Broadcounter/src/shy/luo/broadcounter/Broadcounter.java

```
01 public class Broadcounter extends Activity implements OnClickListener {
02     ......
03
04     @Override
05     public void onResume() {
06         super.onResume();
07
08         IntentFilter counterActionFilter = new IntentFilter(CounterService.BROADCAST_COUNTER_ACTION);
09         registerReceiver(counterActionReceiver, counterActionFilter);
10     }
11
12     ......
13
14     private BroadcastReceiver counterActionReceiver = new BroadcastReceiver(){
15         public void onReceive(Context context, Intent intent) {
16             int counter = intent.getIntExtra(CounterService.COUNTER_VALUE, 0);
17             String text = String.valueOf(counter);
18             counterText.setText(text);
19         }
20     };
21 }
```

Broadcounter组件在内部实现了一个广播接收者counterActionReceiver，用来接收CounterService组件发送出来的广播。

Broadcounter组件在注册广播接收者counterActionReceiver之前，首先会创建一个Action名称为BROADCAST_COUNTER_ACTION的IntentFilter对象，以便可以告诉ActivityManagerService，广播接收

者counterActionReceiver所要接收的广播的类型为BROADCAST_COUNTER_ACTION。

Broadcounter类是从Activity类继承下来的，而后者又继承了ContextWrapper类。ContextWrapper类实现了一个成员函数registerReceiver，用来将一个广播接收者注册到ActivityManagerService中。接下来我们继续分析它的实现。

Step 2：ContextWrapper.registerReceiver

frameworks/base/core/java/android/content/ContextWrapper.java

```
01 public class ContextWrapper extends Context {
02     Context mBase;
03     ......
04
05     @Override
06     public Intent registerReceiver(
07         BroadcastReceiver receiver, IntentFilter filter) {
08         return mBase.registerReceiver(receiver, filter);
09     }
10
11     ......
12 }
```

在前面8.2小节的Step 2中提到，ContextWrapper类的成员变量mBase指向的是一个ContextImpl对象，因此，第8行代码实际上是调用了ContextImpl类的成员函数registerReceiver来注册一个广播接收者。

Step 3：ContextImpl.registerReceiver

frameworks/base/core/java/android/app/ContextImpl.java

```
01 class ContextImpl extends Context {
02     ......
03
04     /*package*/ LoadedApk mPackageInfo;
05     /*package*/ ActivityThread mMainThread;
06     private Context mOuterContext;
07     ......
08
09     @Override
10     public Intent registerReceiver(BroadcastReceiver receiver, IntentFilter filter) {
11         return registerReceiver(receiver, filter, null, null);
12     }
13
14     @Override
15     public Intent registerReceiver(BroadcastReceiver receiver, IntentFilter filter,
16             String broadcastPermission, Handler scheduler) {
17         return registerReceiverInternal(receiver, filter, broadcastPermission,
18             scheduler, getOuterContext());
19     }
20
21     private Intent registerReceiverInternal(BroadcastReceiver receiver,
22             IntentFilter filter, String broadcastPermission,
23             Handler scheduler, Context context) {
24         IIntentReceiver rd = null;
25         if (receiver != null) {
26             if (mPackageInfo != null && context != null) {
27                 if (scheduler == null) {
28                     scheduler = mMainThread.getHandler();
29                 }
30                 rd = mPackageInfo.getReceiverDispatcher(
31                     receiver, context, scheduler,
32                     mMainThread.getInstrumentation(), true);
33             }
34             ......
```

```
35          }
36          try {
37              return ActivityManagerNative.getDefault().registerReceiver(
38                      mMainThread.getApplicationThread(),
39                      rd, filter, broadcastPermission);
40          } catch (RemoteException e) {
41                  return null;
42          }
43      }
44
45      ......
46  }
```

第30行到第32行代码首先调用成员变量mPackageInfo的成员函数getReceiverDispatcher将广播接收者receiver封装成一个实现了IIntentReceiver接口的Binder本地对象rd，接着第37行到第39行代码调用ActivityManagerService代理对象的成员函数registerReceiver将前面获得的Binder本地对象rd，以及IntentFilter对象filter等信息发送给ActivityManagerService，以便ActivityManagerService可以将Binder本地对象rd注册在内部中，以及将IntentFilter对象filter所描述的广播转发给它处理。

在将广播接收者receiver封装成一个实现了IIntentReceiver接口的Binder本地对象rd的过程中，还使用到了另外的两个参数。

第一个参数是通过调用成员变量mMainThread 的成员函数getHandler来获得的。在前面8.3小节的Step 3中提到，ContextImpl类的成员变量mMainThread是用来描述当前应用程序进程的，它的成员函数getHandler的返回值是一个Handler对象，用来向当前应用程序进程的主线程的消息队列发送消息。广播接收者并不是直接获得ActivityManagerService给它发送的广播的。一个广播到达对应的广播接收者所运行在的应用程序进程时，首先会被封装成一个消息，然后再发送到该应用程序进程的主线程的消息队列中，最后才会分发给对应的广播接收者处理。

第二个参数是通过调用成员函数getOuterContext来获得的。在前面8.3小节的Step 3中提到，ContextImpl类的成员函数getOuterContext返回的是其内部的一个成员变量mOuterContext，而ContextImpl类的成员变量mOuterContext指向的是一个Activity组件。在我们这个场景中，这个Activity组件就是Broadcounter组件。这样，前面所获得的Binder本地对象rd就可以知道它所封装的BroadcastReceiver对象receiver是与Broadcounter组件关联在一起的。

接下来，我们继续分析ContextImpl类的成员变量mPackageInfo的成员函数getReceiverDispatcher是如何将BroadcastReceiver对象receiver封装成一个实现了IIntentReceiver接口的Binder本地对象rd的，它的实现如下所示。

frameworks/base/core/java/android/app/LoadedApk.java

```
01 final class LoadedApk {
02     ......
03
04     public IIntentReceiver getReceiverDispatcher(BroadcastReceiver r,
05             Context context, Handler handler,
06             Instrumentation instrumentation, boolean registered) {
07         synchronized (mReceivers) {
08             LoadedApk.ReceiverDispatcher rd = null;
09             HashMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher> map = null;
10             if (registered) {
11                 map = mReceivers.get(context);
12                 if (map != null) {
13                     rd = map.get(r);
14                 }
15             }
16             if (rd == null) {
17                 rd = new ReceiverDispatcher(r, context, handler,
```

```
18                      instrumentation, registered);
19                  if (registered) {
20                      if (map == null) {
21                          map = new HashMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher>();
22                          mReceivers.put(context, map);
23                      }
24                      map.put(r, rd);
25                  }
26              } else {
27                  rd.validate(context, handler);
28              }
29              return rd.getIIntentReceiver();
30          }
31      }
32
33      ......
34  }
```

每一个注册过广播接收者的Activity组件在LoadedApk类中都有一个对应的ReceiverDispatcher对象，它负责将这个被注册的广播接收者与注册它的Activity组件关联在一起。这些ReceiverDispatcher对象保存在一个HashMap中，并且以它们所关联的广播接收者为关键字。最后，用来保存这些ReceiverDispatcher对象的HashMap又以它们所关联的Activity组件的Context接口为关键字保存在LoadedApk类的成员变量mReceivers中。

第8行到第28行代码在LoadedApk类的成员变量mReceivers中检查是否存在一个以广播接收者r为关键字的ReceiverDispatcher对象rd。如果不存在，那么第17行和第18行代码就会首先创建这个ReceiverDispatcher对象rd，接着第24行代码再将它保存在以Context接口context为关键字的HashMap对象map中。如果以Context接口context为关键字的HashMap对象map不存在，那么第21行代码就会先创建它，接着第22行代码再将它保存在LoadedApk类的成员变量mReceivers中。

最后，第29行调用前面所获得的ReceiverDispatcher对象rd的成员函数getIIntentReceiver来获得一个实现了IIntentReceiver接口的Binder本地对象，它的实现如下所示。

frameworks/base/core/java/android/app/LoadedApk.java

```
01 final class LoadedApk {
02      ......
03
04      static final class ReceiverDispatcher {
05          final static class InnerReceiver extends IIntentReceiver.Stub {
06              final WeakReference<LoadedApk.ReceiverDispatcher> mDispatcher;
07              ......
08
09              InnerReceiver(LoadedApk.ReceiverDispatcher rd, boolean strong) {
10                  mDispatcher = new WeakReference<LoadedApk.ReceiverDispatcher>(rd);
11                  ......
12              }
13
14              ......
15          }
16          ......
17
18          final IIntentReceiver.Stub mIIntentReceiver;
19          final BroadcastReceiver mReceiver;
20          final Context mContext;
21          final Handler mActivityThread;
22          ......
23
24          ReceiverDispatcher(BroadcastReceiver receiver, Context context,
```

```
25                  Handler activityThread, Instrumentation instrumentation,
26                  boolean registered) {
27              ......
28
29              mIIntentReceiver = new InnerReceiver(this, !registered);
30              mReceiver = receiver;
31              mContext = context;
32              mActivityThread = activityThread;
33
34              ......
35          }
36          ......
37
38          IIntentReceiver getIIntentReceiver() {
39              return mIIntentReceiver;
40          }
41      }
42
43      ......
44  }
```

ReceiverDispatcher类有一个类型为InnerReceiver的成员变量mIIntentReceiver，它指向一个实现了IIntentReceiver接口的Binder本地对象。

ReceiverDispatcher类还有另外三个成员变量mReceiver、mActivityThread和mContext，其中，成员变量mContext指向了一个Activity组件；成员变量mActivityThread和mReceiver分别指向了与该Activity组件相关联的一个Handler对象和一个广播接收者。例如，Broadcounter组件在注册广播接收者counterActionReceiver时，LoadedApk类就会为它创建一个ReceiverDispatcher对象，这个ReceiverDispatcher对象的成员变量mContext、mReceiver和mActivityThread就分别指向了Broadcounter组件、Broadcounter组件内部的广播接收者counterActionReceiver和Broadcounter组件所运行在的应用程序进程的主线程中的一个Handler对象。

最后，ReceiverDispatcher类的成员函数getIIntentReceiver将其内部的成员变量mIIntentReceiver所指向的一个InnerReceiver对象返回给调用者。

注意

这个InnerReceiver对象内部有一个类型为ReceiverDispatcher的弱引用mDispatcher，它指向了一个外部的ReceiverDispatcher对象。

回到ContextImpl类的成员函数registerReceiverInternal中，它将广播接收者receiver封装成一个InnerReceiver对象之后，最后就将这个InnerReceiver对象注册到ActivityManagerService中了。

Step 4：ActivityManagerProxy.registerReceiver

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
01 class ActivityManagerProxy implements IActivityManager
02 {
03     ......
04
05     public Intent registerReceiver(IApplicationThread caller,
06             IIntentReceiver receiver,
07             IntentFilter filter, String perm) throws RemoteException
08     {
09         Parcel data = Parcel.obtain();
10         Parcel reply = Parcel.obtain();
11         data.writeInterfaceToken(IActivityManager.descriptor);
12         data.writeStrongBinder(caller != null ? caller.asBinder() : null);
13         data.writeStrongBinder(receiver != null ? receiver.asBinder() : null);
```

```
14          filter.writeToParcel(data, 0);
15          data.writeString(perm);
16          mRemote.transact(REGISTER_RECEIVER_TRANSACTION, data, reply, 0);
17          reply.readException();
18          Intent intent = null;
19          int haveIntent = reply.readInt();
20          if (haveIntent != 0) {
21              intent = Intent.CREATOR.createFromParcel(reply);
22          }
23          reply.recycle();
24          data.recycle();
25          return intent;
26      }
27
28      ......
29  }
```

第11行到第15行代码将前面传进来的参数写入到Parcel对象data中，接着再通过ActivityManagerProxy类内部的一个Binder代理对象mRemote向ActivityManagerService发送一个类型为REGISTER_RECEIVER_TRANSACTION的进程间通信请求。

以上4步都是在应用程序Broadcounter中执行的，接下来第5步是在ActivityManagerService中执行的，主要是用来处理 Broadcounter组件发出的类型为REGISTER_RECEIVER_TRANSACTION的进程间通信请求。

Step 5：ActivityManagerService.registerReceiver

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02          implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03      ......
04
05      public Intent registerReceiver(IApplicationThread caller,
06              IIntentReceiver receiver, IntentFilter filter, String permission) {
07          synchronized(this) {
08              ProcessRecord callerApp = null;
09              if (caller != null) {
10                  callerApp = getRecordForAppLocked(caller);
11                  ......
12              }
13
14              List allSticky = null;
15
16              // Look for any matching sticky broadcasts...
17              Iterator actions = filter.actionsIterator();
18              if (actions != null) {
19                  while (actions.hasNext()) {
20                      String action = (String)actions.next();
21                      allSticky = getStickiesLocked(action, filter, allSticky);
22                  }
23              }
24              ......
25
26              // The first sticky in the list is returned directly back to
27              // the client.
28              Intent sticky = allSticky != null ? (Intent)allSticky.get(0) : null;
29              ......
30
```

```
31              ReceiverList rl
32                  = (ReceiverList)mRegisteredReceivers.get(receiver.asBinder());
33              if (rl == null) {
34                  rl = new ReceiverList(this, callerApp,
35                      Binder.getCallingPid(),
36                      Binder.getCallingUid(), receiver);
37
38                  ......
39                  mRegisteredReceivers.put(receiver.asBinder(), rl);
40              }
41
42              BroadcastFilter bf = new BroadcastFilter(filter, rl, permission);
43              rl.add(bf);
44              ......
45              mReceiverResolver.addFilter(bf);
46              ......
47
48              return sticky;
49          }
50      }
51
52      ......
53  }
```

ActivityManagerService类的成员函数registerReceiver用来处理类型为REGISTER_RECEIVER_TRANSACTION的进程间通信请求。

第8行到第12行代码根据参数caller得到一个ProcessRecord对象callerApp，用来描述正在请求ActivityManagerService注册广播接收者的一个Activity组件所运行在的应用程序进程。

第17行到第23行代码检查ActivityManagerService内部是否存在与参数filter所对应的黏性（Sticky）广播。如果存在，那么就将它们保存在列表allSticky中。

注意

参数filter有一个Action列表，用来描述一系列不同类型的广播，但是只有最后面一种类型的黏性广播才会被保存在列表allSticky中。

什么是黏性（Sticky）广播呢？一个黏性广播被发送到ActivityManagerService之后，就会一直保存在ActivityManagerService中，直到ActivityManagerService下次再接收到另外一个同类型的黏性广播为止。一个Activity组件在向ActivityManagerService注册接收某一种类型的广播时，如果ActivityManagerService内部恰好保存有这种类型的黏性广播，那么ActivityManagerService就会将这个粘性广播返回给这个Activity组件，以便它可以知道系统上一次发出的它所感兴趣的广播的内容。我们可以通过调用Activity组件或者Service组件的父类ContextWrapper的成员函数sendStickyBroadcast向ActivityManagerService发送一个黏性广播。

第28行代码检查列表allSticky是否等于null。如果不等于null，那么就说明ActivityManagerService内部存在与参数filter对应的黏性广播，这时候ActivityManagerService就会将列表allSticky中的第一个黏性广播取出来，并且保存在变量sticky中。

注意

广播是使用一个Intent对象来描述的，因此，变量sticky的类型为Intent。

从前面的调用过程可以知道，参数receiver是一个Binder代理对象，引用了一个类型为InnerReceiver

的Binder本地对象。Activity组件在注册一个广播接收者时，并不是真的将这个广播接收者注册到ActivityManagerService中，而是将与它关联的一个InnerReceiver对象注册到ActivityManagerService中。当ActivityManagerService接收到一个广播时，它就会根据这个广播的类型在内部找到对应的InnerReceiver对象，然后再通过此InnerReceiver对象将这个广播发送给对应的广播接收者处理。

在ActivityManagerService中，每一个广播接收者都是使用一个BroadcastFilter对象来描述的，而每一个BroadcastFilter对象又是根据它所描述的广播接收者所关联的一个InnerReceiver对象，以及所要接收的广播的类型来创建的。由于在一个应用程序中，不同的Activity组件可能会使用同一个InnerReceiver对象来注册不同的广播接收者，因此，ActivityManagerService就会使用一个ReceiverList列表来保存这些使用了相同InnerReceiver对象来注册的广播接收者，并且以它们所使用的InnerReceiver对象为关键字。

第31行和第32行代码在ActivityManagerService类的成员变量mRegisteredReceivers中找到与参数receiver所描述的InnerReceiver对象对应的一个ReceiverList列表rl。如果找不到，那么第34行到第36行代码就会先创建这个ReceiverList列表rl，接着第39行代码再将它保存在ActivityManagerService类的成员变量mRegisteredReceivers中。

第42行代码首先根据参数filter、rl和permission创建了一个BroadcastFilter对象bf，用来描述正在注册的广播接收者，接着第43行和第45行代码分别将它添加到前面所获得的ReceiverList列表rl，以及ActivityManagerService类的成员变量mReceiverResolver中。以后ActivityManagerService接收到广播时，它就可以在成员变量mReceiverResolver中找到对应的广播接收者了。

最后，第48行代码将前面所获得的一个黏性广播返回给正在注册广播接收者的一个Activity组件，以便它可以知道系统上一次发出的它所感兴趣的广播的内容。

至此，广播接收者的注册过程就分析完成了，接下来我们继续分析广播的发送过程。

9.3 广播的发送过程

在前面的9.1小节中，应用程序Broadcounter的CounterService组件通过广播将内部的计数器的当前计数值实时地更新到Broadcounter组件的用户界面上。本节将结合这个场景来分析广播的发送过程。

与广播接收者的注册过程相比，广播的发送过程要复杂得多，具体如下所示。

（1）广播发送者，即一个Activity组件或者一个Service组件，将一个特定类型的广播发送给ActivityManagerService。

（2）ActivityManagerService接收到一个广播之后，首先找到与这个广播对应的广播接收者，然后将它们添加到一个广播调度队列中，最后向ActivityManagerService所运行在的线程的消息队列发送一个类型为BROADCAST_INTENT_MSG的消息。这时候对广播发送者来说，一个广播就发送完成了。

（3）当发送到ActivityManagerService所运行在的线程的消息队列中的BROADCAST_INTENT_MSG消息被处理时，ActivityManagerService就会从广播调度队列中找到需要接收广播的广播接收者，并且将对应的广播发送给它们所运行在的应用程序进程。

（4）广播接收者所运行在的应用程序进程接收到ActivityManagerService发送过来的广播之后，并不是直接将接收到的广播分发给广播接收者来处理，而是将接收到的广播封装成一个消息，并且发送到主线程的消息队列中。当这个消息被处理时，应用程序进程才会将它所描述的广播发送给相应的广播接收者处理。

这个过程一共分为17个步骤，接下来我们就详细分析每一个步骤。

第1步到第4步如图9-3所示。

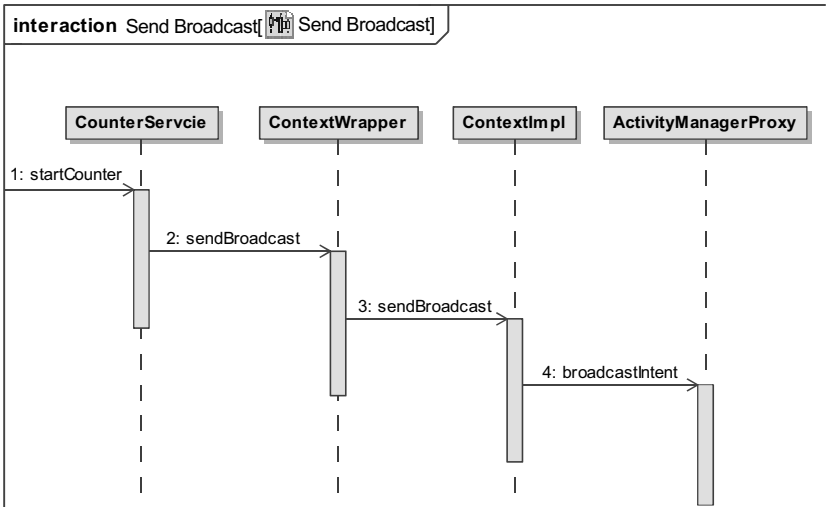


图9-3　广播的发送过程——Step 1至Step 4

Step 1：CounterService.startCounter

packages/experimental/Broadcounter/src/shy/luo/broadcounter/CounterService.java

```
01 public class CounterService extends Service implements ICounterService {
02     ......
03
04     public void startCounter(int initVal) {
05         AsyncTask<Integer, Integer, Integer> task = new AsyncTask<Integer, Integer, Integer>() {
06             ......
07
08             @Override
09             protected void onProgressUpdate(Integer... values) {
10                 super.onProgressUpdate(values);
11
12                 int counter = values[0];
13                 Intent intent = new Intent(BROADCAST_COUNTER_ACTION);
14                 intent.putExtra(COUNTER_VALUE, counter);
15
16                 sendBroadcast(intent);
17             }
18
19             @Override
20             protected void onPostExecute(Integer val) {
21                 int counter = val;
22                 Intent intent = new Intent(BROADCAST_COUNTER_ACTION);
23                 intent.putExtra(COUNTER_VALUE, counter);
24
25                 sendBroadcast(intent);
26            }
27
28         };
29
30         task.execute(initVal);
31     }
32
33     ......
34 }
```

CounterService组件在内部通过一个异步任务（AsyncTask）来实现一个计数器功能。在计数器的执行过程中，或者计数器结束计数时，CounterService组件都会将计数器的当前计数值写入到一个类型为BROADCAST_COUNTER_ACTION的广播中，并且将这个广播发送到ActivityManagerService中，以便可以将这个广播发送给Broadcounter组件。

注意

一个广播是使用一个Intent对象来描述的，而这个Intent对象的Action名称就用来描述它所对应的广播的类型。

CounterService组件是通过调用其父类ContextWrapper的成员函数sendBroadcast来向ActivityManagerService发送一个广播的，因此，接下来我们就分析ContextWrapper类的成员函数sendBroadcast的实现。

Step 2：ContextWrapper.sendBroadcast

frameworks/base/core/java/android/content/ContextWrapper.java

```
01 public class ContextWrapper extends Context {
02     Context mBase;
03     ......
04
05     @Override
06     public void sendBroadcast(Intent intent) {
07         mBase.sendBroadcast(intent);
08     }
09
10     ......
11 }
```

在前面8.2小节的Step 2中提到，ContextWrapper类的成员变量mBase指向的是一个ContextImpl对象，因此，第7行代码实际上是调用了ContextImpl类的成员函数sendBroadcast来发送一个广播。

Step 3：ContextImpl.sendBroadcast

frameworks/base/core/java/android/app/ContextImpl.java

```
01 class ContextImpl extends Context {
02     ......
03
04     @Override
05     public void sendBroadcast(Intent intent) {
06         String resolvedType = intent.resolveTypeIfNeeded(getContentResolver());
07         try {
08             ActivityManagerNative.getDefault().broadcastIntent(
09                 mMainThread.getApplicationThread(), intent, resolvedType, null,
10                 Activity.RESULT_OK, null, null, null, false, false);
11         } catch (RemoteException e) {
12         }
13     }
14
15     ......
16 }
```

第8行和第10行代码首先调用ActivityManagerNative类的静态成员函数getDefault来获得ActivityManagerService的一个代理对象，然后调用这个代理对象的成员函数broadcastIntent将参数intent

所描述的一个广播发送给ActivityManagerService，同时发送给ActivityManagerService的还有当前应用程序进程中的一个ApplicationThread对象，以便ActivityManagerService可以知道是谁向它发送了一个广播。

ActivityManagerService代理对象的类型为ActivityManagerProxy，因此，接下来我们就分析ActivityManagerProxy类的成员函数broadcastIntent的实现。

Step 4：ActivityManagerProxy.broadcastIntent

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
01 class ActivityManagerProxy implements IActivityManager
02 {
03     ......
04
05     public int broadcastIntent(IApplicationThread caller,
06         Intent intent, String resolvedType,  IIntentReceiver resultTo,
07         int resultCode, String resultData, Bundle map,
08         String requiredPermission, boolean serialized,
09         boolean sticky) throws RemoteException
10     {
11         Parcel data = Parcel.obtain();
12         Parcel reply = Parcel.obtain();
13         data.writeInterfaceToken(IActivityManager.descriptor);
14         data.writeStrongBinder(caller != null ? caller.asBinder() : null);
15         intent.writeToParcel(data, 0);
16         data.writeString(resolvedType);
17         data.writeStrongBinder(resultTo != null ? resultTo.asBinder() : null);
18         data.writeInt(resultCode);
19         data.writeString(resultData);
20         data.writeBundle(map);
21         data.writeString(requiredPermission);
22         data.writeInt(serialized ? 1 : 0);
23         data.writeInt(sticky ? 1 : 0);
24         mRemote.transact(BROADCAST_INTENT_TRANSACTION, data, reply, 0);
25         reply.readException();
26         int res = reply.readInt();
27         reply.recycle();
28         data.recycle();
29         return res;
30     }
31
32     ......
33 }
```

第13行到第23行代码将前面传进来的参数写入到Parcel对象data中，接着再通过ActivityManagerProxy类内部的一个Binder代理对象mRemote向ActivityManagerService发送一个类型为BROADCAST_INTENT_TRANSACTION的进程间通信请求。

以上4步都是在应用程序Broadcounter中执行的，接下来第5步到第12步是在ActivityManagerService中执行的，主要是用来处理 CounterService组件发出的类型为BROADCAST_INTENT_TRANSACTION的进程间通信请求，如图9–4所示。

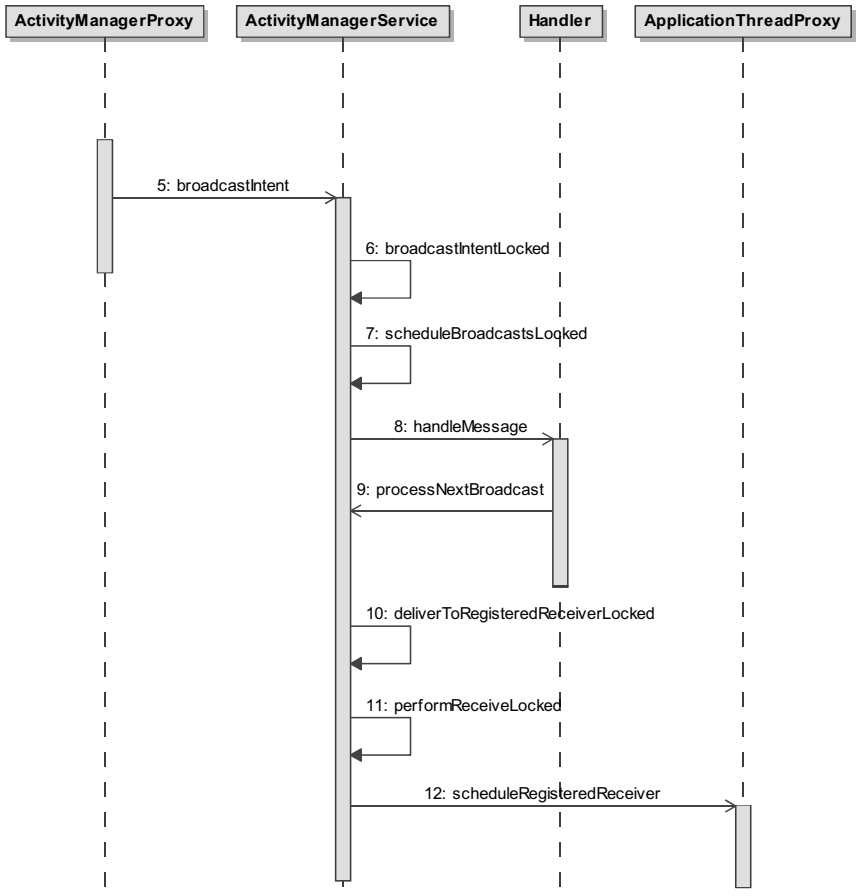


图9–4 广播的发送过程——Step 5至Step 12

Step 5：ActivityManagerService.broadcastIntent

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     public final int broadcastIntent(IApplicationThread caller,
06             Intent intent, String resolvedType, IIntentReceiver resultTo,
07             int resultCode, String resultData, Bundle map,
08             String requiredPermission, boolean serialized, boolean sticky) {
09         synchronized(this) {
10             intent = verifyBroadcastLocked(intent);
11
12             final ProcessRecord callerApp = getRecordForAppLocked(caller);
13             final int callingPid = Binder.getCallingPid();
14             final int callingUid = Binder.getCallingUid();
15             final long origId = Binder.clearCallingIdentity();
16             int res = broadcastIntentLocked(callerApp,
17                 callerApp != null ? callerApp.info.packageName : null,
18                 intent, resolvedType, resultTo,
19                 resultCode, resultData, map, requiredPermission, serialized,
20                 sticky, callingPid, callingUid);
21             Binder.restoreCallingIdentity(origId);
22             return res;
23         }
24     }
25
26     ......
27 }
```

ActivityManagerService类的成员函数broadcastIntent用来处理类型为BROADCAST_INTENT_TRANSACTION的进程间通信请求。

第10行调用成员函数verifyBroadcastLocked来验证参数intent所描述的广播的内容是否合法，例如，它是否携带有文件描述符。此外，如果这时候系统正处于启动的过程中，那么成员函数verifyBroadcastLocked还会验证参数intent所描述的广播是否只发送给动态注册的广播接收者。在系统的启动过程中，PackageManagerService可能还未被启动，在这种情况下，ActivityManagerService是无法获得静态注册的广播接收者的，因此，就禁止发送广播给静态注册的广播接收者。

第12行到第15行代码首先获得发送广播的应用程序进程的身份，接着第16行到第20行代码调用成员函数broadcastIntentLocked来处理参数intent所描述的广播。

Step 6：ActivityManagerService.broadcastIntentLocked

这一步主要是用来查找目标广播接收者的，我们分段来阅读。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
001 public final class ActivityManagerService extends ActivityManagerNative
002         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
003     ......
004
005     private final int broadcastIntentLocked(ProcessRecord callerApp,
006             String callerPackage, Intent intent, String resolvedType,
007             IIntentReceiver resultTo, int resultCode, String resultData,
008             Bundle map, String requiredPermission,
009             boolean ordered, boolean sticky, int callingPid, int callingUid) {
010         intent = new Intent(intent);
011         ......
012
013         // Add to the sticky list if requested.
014         if (sticky) {
015             ......
016
017             ArrayList<Intent> list = mStickyBroadcasts.get(intent.getAction());
018             if (list == null) {
019                 list = new ArrayList<Intent>();
020                 mStickyBroadcasts.put(intent.getAction(), list);
021             }
022             int N = list.size();
023             int i;
024             for (i=0; i<N; i++) {
025                 if (intent.filterEquals(list.get(i))) {
026                     // This sticky already exists, replace it.
027                     list.set(i, new Intent(intent));
028                     break;
029                 }
030             }
031             if (i >= N) {
032                 list.add(new Intent(intent));
033             }
034         }
```

参数sticky用来描述参数intent所描述的广播是否是一个黏性广播。如果是，它的值就等于true，即14行的if语句为true。这时候ActivityManagerService就需要将这个广播保存下来，以便后面注册要接收此种类型广播的广播接收者可以获得这个广播。

在ActivityManagerService类中，所有相同类型的黏性广播都保存在一个列表中，这些列表最终保存在ActivityManagerService类的成员变量mStickyBroadcasts所描述的一个HashMap中，并且以它们的广播类型为关键字。

前面提到，一个广播是使用一个Intent对象来描述的，而广播的类型则是由这个Intent对象的Action名称来描述的。因此，第17行代码就在ActivityManagerService类的成员变量mStickyBroadcasts中查找是否存在与参数intent的Action名称对应的一个黏性广播列表list。如果不存在，那么第19行代码就会先创建这个黏性广播列表list，并且第20行代码将它保存在成员变量mStickyBroadcasts中。

第24行到第30行的for循环检查在黏性广播列表list中是否存在一个与参数intent一致的广播。如果存在，那么第27行代码就用参数intent所描述的广播来替换它；否则，第32行代码就将参数intent所描述的广播添加到黏性广播列表list中。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
035         // Figure out who all will receive this broadcast.
036         List receivers = null;
037         List<BroadcastFilter> registeredReceivers = null;
038         try {
039             if (intent.getComponent() != null) {
040                 // Broadcast is going to one specific receiver class...
041                 ActivityInfo ai = AppGlobals.getPackageManager().
042                     getReceiverInfo(intent.getComponent(), STOCK_PM_FLAGS);
043                 if (ai != null) {
044                     receivers = new ArrayList();
045                     ResolveInfo ri = new ResolveInfo();
046                     ri.activityInfo = ai;
047                     receivers.add(ri);
048                 }
049             } else {
050                 // Need to resolve the intent to interested receivers...
051                 if ((intent.getFlags()&Intent.FLAG_RECEIVER_REGISTERED_ONLY)
052                           == 0) {
053                     receivers =
054                         AppGlobals.getPackageManager().queryIntentReceivers(
055                                 intent, resolvedType, STOCK_PM_FLAGS);
056                 }
057                 registeredReceivers = mReceiverResolver.queryIntent(intent, resolvedType, false);
058             }
059         } catch (RemoteException ex) {
060             ......
061         }
```

如果参数intent指定了广播接收者的组件名称，即第39行的if语句为true，那么就说明参数intent所描述的广播是要发送给特定的广播接收者的，因此，第41行和第42行代码就首先根据参数intent所指定的组件名称在PackageManagerService中找到对应的广播接收者，接着第44行到第47行代码再将找到的广播接收者保存在列表receivers中。

如果参数intent没有指定广播接收者的组件名称，即第39行的if语句为false，那么就说明参数intent所描述的广播是要发送给所有注册的，并且要接收它的广播接收者的。这些目标广播接收者分为两种类型：一种是静态注册的，另外一种是动态注册的。如果参数intent的标志位FLAG_RECEIVER_REGISTERED_ONLY等于1，即第51行和第52行的if语句为false，那么它所描述的广播就只可以发送给动态注册的广播接收者；否则，第53行到第55行代码就会同时到PackageManagerService中找到所有静态注册的目标广播接收者，并且保存在列表receivers中。在这里就可以看出，无论如何，在这种情况下，参数intent所描述的广播都是需要发送给动态注册的目标广播接收者的。从前面9.2节的内容可以知道，所有动态注册的广播接收者都是保存在ActivityManagerService类的成员变量mReceiverResolver中的，因此，接下来第57行代码就在里面找到动态注册的目标广播接收者，并且将它们保存在另外一个列表registeredReceivers中。

这样，ActivityManagerService就得到了所有的目标广播接收者，其中，属于静态注册的目标广播接收者保存在列表receivers中，而属于动态注册的目标广播接收者保存在列表registeredReceivers中。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
062         final boolean replacePending =
063             (intent.getFlags()&Intent.FLAG_RECEIVER_REPLACE_PENDING) != 0;
064
065         int NR = registeredReceivers != null ? registeredReceivers.size() : 0;
066         if (!ordered && NR > 0) {
067             // If we are not serializing this broadcast, then send the
068             // registered receivers separately so they don't wait for the
069             // components to be launched.
070             BroadcastRecord r = new BroadcastRecord(intent, callerApp,
071                 callerPackage, callingPid, callingUid, requiredPermission,
072                 registeredReceivers, resultTo, resultCode, resultData, map,
073                 ordered, sticky, false);
074             ......
075             boolean replaced = false;
076             if (replacePending) {
077                 for (int i=mParallelBroadcasts.size()-1; i>=0; i--) {
078                     if (intent.filterEquals(mParallelBroadcasts.get(i).intent)) {
079                         ......
080                         mParallelBroadcasts.set(i, r);
081                         replaced = true;
082                         break;
083                     }
084                 }
085             }
086
087             if (!replaced) {
088                 mParallelBroadcasts.add(r);
089                 scheduleBroadcastsLocked();
090             }
091
092             registeredReceivers = null;
093             NR = 0;
094         }
```

由于ActivityManagerService是通过消息处理机制将接收到的广播转发给目标广播接收者的，因此，就可能会出现这样一种情况：上次接收的一个广播还未来得及转发给目标广播接收者，又马上接收到一个同样广播。在这种情况下，如果现在接收的广播的标志位FLAG_RECEIVER_REPLACE_PENDING等于1，那么ActivityManagerService就会使用新的广播来替换旧的广播。因此，第62行和第63行代码就会检查参数intent所描述的广播的标志位FLAG_RECEIVER_REPLACE_PENDING是否等于1。如果等于1，就将变量replacePending的值设置为true。

参数ordered用来描述当前发送的广播是否是一个无序广播。如果是，并且存在动态注册的目标广播接收者，即第66行的if语句为true，那么第70行到第93行代码就会首先将当前发送的广播，即参数intent所描述的广播，转发给这些动态注册的目标广播接收者，然后再转发给静态注册的目标广播接收者。从这里就可以看出，动态注册的广播接收者要比静态注册的广播接收者优先接收到无序广播。

第70行到第73行代码将参数intent所描述的广播，以及动态注册的目标广播接收者封装成一个BroadcastRecord对象r，用来描述ActivityManagerService要执行的一个广播转发任务。

注意

这个广播转发任务并不是马上执行的，而是被添加到ActivityManagerService内部的一个无序广播调度队列中去等待执行，这个无序广播调度队列是使用成员变量mParallelBroadcasts来描述的。

在将BroadcastRecord对象r所描述的广播转发任务添加到ActivityManagerService的无序广播调度队列之前，如果变量replacePending的值等于true，即第76行的if语句为true，那么第77行到第84行的for循

环就会在这个无序广播调度队列中检查是否存在这样的一个广播，它与参数intent所描述的广播是一致的。如果存在，那么第80行代码就会使用参数intent所描述的广播来替换它，并且第81行代码将变量replaced的值设置为true。

第87行的if语句检查变量replaced的值是否等于true。如果是，那么就说明不需要在ActivityManagerService的无序广播调度队列中增加新的广播转发任务，因此，这时候就不需要重复去调度这个队列中的广播转发任务；否则，第88行代码就会将BroadcastRecord对象r所描述的广播转发任务添加到ActivityManagerService的无序广播调度队列中，并且第89行调用成员函数scheduleBroadcastsLocked来重新调度这个队列中的广播转发任务。

这样，对于无序广播来说，ActivityManagerService就相当于已经将参数intent所描述的广播转发给那些动态注册的目标广播接收者了。因此，第92行代码就将列表registeredReceivers的值设置为null，并且第93行代码将用来描述动态注册的目标广播接收者的个数的变量NR的值设置为0。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
095         // Merge into one list.
096         int ir = 0;
097         if (receivers != null) {
098             ......
099
100             int NT = receivers != null ? receivers.size() : 0;
101             int it = 0;
102             ResolveInfo curt = null;
103             BroadcastFilter curr = null;
104             while (it < NT && ir < NR) {
105                 if (curt == null) {
106                     curt = (ResolveInfo)receivers.get(it);
107                 }
108                 if (curr == null) {
109                     curr = registeredReceivers.get(ir);
110                 }
111                 if (curr.getPriority() >= curt.priority) {
112                     // Insert this broadcast record into the final list.
113                     receivers.add(it, curr);
114                     ir++;
115                     curr = null;
116                     it++;
117                     NT++;
118                 } else {
119                     // Skip to the next ResolveInfo in the final list.
120                     it++;
121                     curt = null;
122                 }
123             }
124         }
125         while (ir < NR) {
126             if (receivers == null) {
127                 receivers = new ArrayList();
128             }
129             receivers.add(registeredReceivers.get(ir));
130             ir++;
131         }
```

函数执行到这里时，无论ActivityManagerService当前接收到的是一个无序广播，还是一个有序广播，ActivityManagerService都会将该广播及其目标广播接收者封装成一个广播转发任务，并且添加到内部的一个有序广播调度队列中。但是对于无序广播来说，当它们真正被转发时，会得到特殊处理，即不按照有序广播的方式来转发。

ActivityManagerService在内部使用成员变量mOrderedBroadcasts来描述一个有序广播调度队列，队

列中的每一个广播转发任务的目标广播接收者都是按照优先级从高到低的顺序来排列的，不管它们是动态注册的，还是静态注册的。

第104行到第123行的while循环用来合并动态注册和静态注册的目标广播接收者。合并后得到的目标广播接收者都保存在列表receivers中，并且它们是按照优先级从高到低的顺序来排列的。

注意

在合并之前，保存在registeredReceivers和receivers两个列表中的目标广播接收者都是按照优先级从高到低的顺序来排列的，因此，在合并这两个列表时，就会充分利用这个特性，使得合并后的目标广播接收者仍然按照优先级从高到低的顺序排列在列表receivers中。从第111行的if语句可以看出，在合并的过程中，如果一个动态注册的目标广播接收者和一个静态注册的目标广播接收者的优先级相同，那么动态注册的目标广播接收者会排在静态注册的目标广播接收者的前面，即动态注册的目标广播接收者会优先于静态注册的目标广播接收者接收到有序广播。

第104行到第123行的while循环执行完成之后，可能还没有完全将registeredReceivers列表中的目标广播接收者合并到列表receivers中。在这种情况下，第125行到第131行的while循环就需要依次将它们添加到列表receivers的末尾，因为它们是优先级最低的目标广播接收者。

这样，对于无序广播来说，静态注册的目标广播接收者就全部保存在列表receivers中了；而对于有序广播来说，静态注册和动态注册的目标广播接收者也全部保存在列表receivers中了。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
132         if ((receivers != null && receivers.size() > 0)
133                 || resultTo != null) {
134             BroadcastRecord r = new BroadcastRecord(intent, callerApp,
135                     callerPackage, callingPid, callingUid, requiredPermission,
136                     receivers, resultTo, resultCode, resultData, map, ordered,
137                     sticky, false);
138             ......
139             boolean replaced = false;
140             if (replacePending) {
141                 for (int i=mOrderedBroadcasts.size()-1; i>0; i--) {
142                     if (intent.filterEquals(mOrderedBroadcasts.get(i).intent)) {
143                         ......
144                         mOrderedBroadcasts.set(i, r);
145                         replaced = true;
146                         break;
147                     }
148                 }
149             }
150
151             if (!replaced) {
152                 mOrderedBroadcasts.add(r);
153                 scheduleBroadcastsLocked();
154             }
155         }
156
157         return BROADCAST_SUCCESS;
158     }
159
160     ......
161 }
```

第132行到第155行代码将参数intent所描述的广播，以及剩余的其他目标广播接收者封装成另外一个BroadcastRecord对象r，用来描述ActivityManagerService要执行的另一个广播转发任务，并且添加到ActivityManagerService内部的有序广播调度队列中。

注意

在添加之后，如果变量replacePending的值等于true，即第140行的if语句为true，那么第141行到第148行的for循环就会在这个有序广播调度队列中检查是否存在这样的一个广播，它与参数intent所描述的广播是一致的。如果存在，那么第144行代码就会使用参数intent所描述的广播来替换它，并且第145行代码将变量replaced的值设置为true。

第151行的if语句检查变量replaced的值是否等于true。如果是，那么就说明不需要在ActivityManagerService的有序广播调度队列中增加新的广播转发任务，因此，这时候就不需要重复去调度这个队列中的广播转发任务；否则，第152行代码就会将BroadcastRecord对象r所描述的广播转发任务添加到ActivityManagerService的有序广播调度队列中，并且第153行代码调用成员函数scheduleBroadcastsLocked来重新调度这个队列中的广播转发任务。

至此，ActivityManagerService就找到参数intent所描述的广播的目标广播接收者了，并且分别将它们保存在内部的无序广播调度队列mParallelBroadcasts和有序广播调度队列mOrderedBroadcasts中。接下来，ActivityManagerService就会继续调用成员函数scheduleBroadcastsLocked将参数intent所描述的广播转发给目标广播接收者处理。

Step 7：ActivityManagerService.scheduleBroadcastsLocked

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     private final void scheduleBroadcastsLocked() {
06         ......
07
08         if (mBroadcastsScheduled) {
09             return;
10         }
11
12         mHandler.sendEmptyMessage(BROADCAST_INTENT_MSG);
13         mBroadcastsScheduled = true;
14     }
15
16     ......
17 }
```

ActivityManagerService类有一个成员变量mBroadcastsScheduled，用来描述ActivityManagerService是否已经向它所运行在的线程的消息队列发送了一个类型为BROADCAST_INTENT_MSG的消息。ActivityManagerService就是通过这个BROADCAST_INTENT_MSG消息来调度保存在无序广播调度队列mParallelBroadcasts和有序广播调度队列mOrderedBroadcasts中的广播转发任务的。

第8行的if语句首先检查成员变量mBroadcastsScheduled的值是否等于true。如果是，那么就说明ActivityManagerService所运行在的线程的消息队列中已经存在一个类型为BROADCAST_INTENT_MSG的消息了；否则，第12行代码就向这个线程的消息队列发送一个类型为BROADCAST_INTENT_MSG的消息，并且第13行代码将成员变量mBroadcastsScheduled的值设置为true。这个消息最终是在ActivityManagerService类的成员变量mHandler的成员函数handleMessage中处理的。

注意

虽然ActivityManagerService还没有将广播转发给目标广播接收者，但是当它执行完成这一步之后，广播发送者就会认为这个广播已经成功地发送出去了。从这里就可以看出，广播的

发送和接收是异步的，即广播发送者将一个广播发送给ActivityManagerService之后，不会等待ActivityManagerService将这个广播转发给广播接收者处理。

Step 8：Handler.handle Message

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     final Handler mHandler = new Handler() {
06         public void handleMessage(Message msg) {
07             switch (msg.what) {
08             ......
09             case BROADCAST_INTENT_MSG: {
10                 ......
11                 processNextBroadcast(true);
12             } break;
13             ......
14             }
15         }
16     }
17
18     ......
19 }
```

第11行调用ActivityManagerService类的成员函数processNextBroadcast来处理类型为BROADCAST_INTENT_MSG的消息，用来调度保存在ActivityManagerService内部的无序广播调度队列mParallelBroadcasts和有序广播调度队列mOrderedBroadcasts中的广播转发任务。

Step 9：ActivityManagerService.processNextBroadcast

这一步主要是用来将广播转发给目标广播接收者处理，我们分段来阅读。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
001 public final class ActivityManagerService extends ActivityManagerNative
002         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
003     ......
004
005     private final void processNextBroadcast(boolean fromMsg) {
006         synchronized(this) {
007             BroadcastRecord r;
008             ......
009
010             if (fromMsg) {
011                 mBroadcastsScheduled = false;
012             }
013
014             // First, deliver any non-serialized broadcasts right away.
015             while (mParallelBroadcasts.size() > 0) {
016                 r = mParallelBroadcasts.remove(0);
017                 ......
018                 final int N = r.receivers.size();
019                 ......
020                 for (int i=0; i<N; i++) {
021                     Object target = r.receivers.get(i);
022                     ......
023
024                     deliverToRegisteredReceiverLocked(r, (BroadcastFilter)target, false);
025                 }
026                 ......
027             }
```

参数fromMsg用来描述ActivityManagerService类的成员函数processNextBroadcast是否是用来处理类型为BROADCAST_INTENT_MSG的消息的。如果是，即第10行的if语句为true，那么第11行就将ActivityManagerService类的成员变量mBroadcastsScheduled的值设置为false，表示前面发送到ActivityManagerService所运行在的线程的消息队列中的BROADCAST_INTENT_MSG消息已经被处理了。

第15行到第27行的while循环用来处理保存在无序广播调度队列mParallelBroadcasts中的广播转发任务，即将保存在无序广播调度队列mParallelBroadcasts中的广播发送给它的目标广播接收者处理。第16行代码首先得到保存在无序广播调度队列mParallelBroadcasts中的每一个BroadcastRecord对象r，接着第20行到第25行的for循环再调用成员函数deliverToRegisteredReceiverLocked将它所描述的一个无序广播发送给每一个目标广播接收者。

接下来，ActivityManagerService就继续处理保存在有序广播调度队列mOrderedBroadcasts中的广播转发任务。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
028             // Now take care of the next serialized one...
029
030             // If we are waiting for a process to come up to handle the next
031             // broadcast, then do nothing at this point.  Just in case, we
032             // check that the process we're waiting for still exists.
033             if (mPendingBroadcast != null) {
034                 ......
035
036                 boolean isDead;
037                 synchronized (mPidsSelfLocked) {
038                     isDead = (mPidsSelfLocked.get(mPendingBroadcast.curApp.pid) == null);
039                 }
040                 if (!isDead) {
041                     // It's still alive, so keep waiting
042                     return;
043                 } else {
044                     ......
045                     mPendingBroadcast.state = BroadcastRecord.IDLE;
046                     mPendingBroadcast.nextReceiver = mPendingBroadcastRecvIndex;
047                     mPendingBroadcast = null;
048                 }
049             }
```

从前面的Step 6可以知道，有序广播调度队列mOrderedBroadcasts描述的广播的目标广播接收者有可能是静态注册的，而这些静态注册的目标广播接收者可能还没有被启动起来，因此，ActivityManagerService将一个广播发送给它们处理时，首先要将它们启动起来。事实上，ActivityManagerService只需要将它们所运行在的应用程序进程启动起来就行了，因为当这些应用程序进程接收到ActivityManagerService发送过来的广播之后，就会主动将目标广播接收者启动起来。

ActivityManagerService类的成员变量mPendingBroadcast就是用来描述一个正在等待静态注册的目标广播接收者启动起来的广播转发任务的。如果存在这样的广播转发任务，即第33行的if语句为true，那么第36行到第48行代码就会检查这个静态注册的目标广播接收者所运行在的应用程序进程是否已经启动起来。如果这个应用程序进程正在启动，即第40行的if语句为true，那么ActivityManagerService就会继续等待；否则，第45行到第47行代码就会准备向它发送一个广播。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
050             do {
051                 if (mOrderedBroadcasts.size() == 0) {
052                     ......
053                     return;
054                 }
```

```
055                 r = mOrderedBroadcasts.get(0);
056                 boolean forceReceive = false;
057                 ......
058
059                 int numReceivers = (r.receivers != null) ? r.receivers.size() : 0;
060                 if (mProcessesReady && r.dispatchTime > 0) {
061                     long now = SystemClock.uptimeMillis();
062                     if ((numReceivers > 0) &&
063                             (now > r.dispatchTime + (2*BROADCAST_TIMEOUT*numReceivers))) {
064                         ......
065                         broadcastTimeoutLocked(false); // forcibly finish this broadcast
066                         forceReceive = true;
067                         r.state = BroadcastRecord.IDLE;
068                     }
069                 }
070
071                 if (r.state != BroadcastRecord.IDLE) {
072                     ......
073                     return;
074                 }
075
076                 if (r.receivers == null || r.nextReceiver >= numReceivers
077                         || r.resultAbort || forceReceive) {
078                     // No more receivers for this broadcast!  Send the final
079                     // result if requested...
080                     ......
081
082                     cancelBroadcastTimeoutLocked();
083                     ......
084
085                     // ... and on to the next...
086                     ......
087                     mOrderedBroadcasts.remove(0);
088                     r = null;
089                     ......
090                     continue;
091                 }
092             } while (r == null);
```

第50行到第92行的while循环在有序广播调度队列mOrderedBroadcasts中找到下一个需要处理的广播转发任务。

第51行的if语句首先检查有序广播调度队列mOrderedBroadcasts是否等于空。如果等于空，那么就说明保存在有序广播调度队列mOrderedBroadcasts中的广播转发任务已经处理完成了；否则，下一个需要处理的广播转发任务就保存在有序广播调度队列mOrderedBroadcasts的第一个位置上，因此，第55行代码就将它取出来，并且保存在BroadcastRecord对象r中。

第59行代码得到BroadcastRecord对象r所描述的广播转发任务的目标广播接收者的个数，并且保存在变量numReceivers中，接着第60行到第69行代码检查前一个目标广播接收者是否在规定的时间内处理完成ActivityManagerService给它发送的一个有序广播。ActivityManagerService在处理BroadcastRecord对象r所描述的一个广播转发任务时，会将当前时间记录在这个BroadcastRecord对象r的成员变量dispatchTime中。如果这个广播转发任务不能在（2*BROADCAST_TIMEOUT*numReceivers）毫秒内处理完成，即它的目标广播接收者不能在（2*BROADCAST_TIMEOUT*numReceivers）毫秒内处理完成ActivityManagerService给它们发送的一个有序广播，那么第65行就会先调用成员函数broadcastTimeoutLocked来强制结束这个广播转发任务，接着第66行和第67行代码再分别将变量forceReceive和BroadcastRecord对象r的成员变量state的值设置为true和BroadcastRecord.IDLE，表示要继续处理有序广播调度队列mOrderedBroadcasts中的下一个广播转发任务。

第71行的if语句检查BroadcastRecord对象r所描述的广播转发任务是否正在处理中，即ActivityManagerService正在将一个有序广播转发给它的前一个目标广播接收者处理。如果是，那么ActivityManagerService就需要等待这个目标广播接收者处理完成该有序广播，然后再转发给下一个目标广播接收者处理，因此，第73行代码就直接返回了。

第76行和第77行的if语句检查BroadcastRecord对象r所描述的广播转发任务是否已经处理完成了，或者它是否已经被强制结束了。如果是，那么第82行就会调用成员函数cancelBroadcastTimeoutLocked来删除前面发送到ActivityManagerService所运行在的线程的消息队列中的一个BROADCAST_TIMEOUT_MSG消息，表示BroadcastRecord对象r所描述的广播转发任务已经在规定的时间内处理完成了。接下来第87行代码就将BroadcastRecord对象r所描述的广播转发任务从有序广播调度队列mOrderedBroadcasts中删除，并且第88行代码将BroadcastRecord对象r的值设置为null，以便可以重复执行第50行到第92行的while循环，来找到下一个需要处理的广播转发任务。

跳出了第50行到第92行的while循环之后，下一个需要处理的广播转发任务就保存在BroadcastRecord对象r中了，接下来，ActivityManagerService就对它进行处理。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
093             // Get the next receiver...
094             int recIdx = r.nextReceiver++;
095
096             // Keep track of when this receiver started, and make sure there
097             // is a timeout message pending to kill it if need be.
098             r.receiverTime = SystemClock.uptimeMillis();
099             if (recIdx == 0) {
100                 r.dispatchTime = r.receiverTime;
101                 ......
102             }
103            if (! mPendingBroadcastTimeoutMessage) {
104                 long timeoutTime = r.receiverTime + BROADCAST_TIMEOUT;
105                 setBroadcastTimeoutLocked(timeoutTime);
106             }
107
108             Object nextReceiver = r.receivers.get(recIdx);
109             if (nextReceiver instanceof BroadcastFilter) {
110                  // Simple case: this is a registered receiver who gets
111                  // a direct call.
112                  BroadcastFilter filter = (BroadcastFilter)nextReceiver;
113                  ......
114                  deliverToRegisteredReceiverLocked(r, filter, r.ordered);
115                  if (r.receiver == null || !r.ordered) {
116                       // The receiver has already finished, so schedule to
117                       // process the next one.
118                       ......
119                       r.state = BroadcastRecord.IDLE;
120                       scheduleBroadcastsLocked();
121                   }
122                  return;
123             }
```

BroadcastRecord对象r所描述的广播转发任务的目标广播接收者保存在它的成员变量receivers所描述的一个列表中，而下一个目标广播接收者就保存在它的成员变量nextReceiver中。

第94行代码首先得到BroadcastRecord对象r所描述的广播转发任务的下一个目标广播接收者在其目标广播接收者列表中的位置，并且保存在变量recIdx中。如果变量recIdx的值等于0，即第99行的if语句为true，那么就说明BroadcastRecord对象r所描述的广播转发任务刚开始被处理，因此，第100行代码就将当前时间记录在它的成员变量dispatchTime中。

第103行的if语句检查ActivityManagerService是否已经向它所运行在的线程的消息队列发送了

一个类型为BROADCAST_TIMEOUT_MSG的消息。如果还没有发送，那么第105行就会调用成员函数setBroadcastTimeoutLocked向这个线程的消息队列发送一个类型为BROADCAST_TIMEOUT_MSG的消息，并且指定它在BROADCAST_TIMEOUT毫秒后处理。在前面的第98行代码中，ActivityManagerService将当前时间保存在了BroadcastRecord对象r的成员变量receiverTime中，表示ActivityManagerService将一个有序广播发送给BroadcastRecord对象r所描述的广播转发任务的下一个目标广播接收者处理的时间。如果这个广播接收者不能在BROADCAST_TIMEOUT毫秒之内处理完成这个有序广播，那么ActivityManagerService就会认为它处理超时了。

第108行代码得到BroadcastRecord对象r所描述的广播转发任务的下一个目标广播接收者nextReceiver。从前面的Step 6可以知道，如果下一个目标广播接收者nextReceiver的类型为BroadcastFilter，那么就说明它是一个动态注册的广播接收者。在这种情况下，第114行就可以直接调用成员函数deliverToRegisteredReceiverLocked向它发送一个广播，因为动态注册的广播接收者肯定是已经启动起来了。接下来第115行的if语句检查BroadcastRecord对象r所描述的广播转发任务是否是用来转发无序广播的。如果是，那么第119行代码就可以直接将BroadcastRecord对象r的成员变量state的值设置为BroadcastRecord.IDLE，表示ActivityManagerService不需要等待它的前一个目标广播接收者处理完成一个广播，就可以将该广播继续发送给它的下一个目标广播接收者处理。接下来第120行调用成员函数scheduleBroadcastsLocked就是为了将一个广播继续发送给BroadcastRecord对象r所描述的广播转发任务的下一个目标广播接收者处理的。

如果BroadcastRecord对象r所描述的广播转发任务的下一个目标广播接收者nextReceiver的类型不是BroadcastFilter，那么就说明这是一个静态注册的广播接收者。这种情况的处理就要复杂一些，因为静态注册的广播接收者可能还没有被启动起来。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
124             // Hard case: need to instantiate the receiver, possibly
125             // starting its application process to host it.
126
127             ResolveInfo info =
128                 (ResolveInfo)nextReceiver;
129             ......
130
131             String targetProcess = info.activityInfo.processName;
132             ......
133
134             // Is this receiver's application already running?
135             ProcessRecord app = getProcessRecordLocked(targetProcess,
136                     info.activityInfo.applicationInfo.uid);
137             if (app != null && app.thread != null) {
138                 try {
139                     processCurBroadcastLocked(r, app);
140                     return;
141                 } catch (RemoteException e) {
142                     ......
143                 }
144                 ......
145             }
146
147             // Not running -- get it started, to be executed when the app comes up.
148             ......
149             if ((r.curApp=startProcessLocked(targetProcess,
150                     info.activityInfo.applicationInfo, true,
151                     r.intent.getFlags() | Intent.FLAG_FROM_BACKGROUND,
152                     "broadcast", r.curComponent,
153                     (r.intent.getFlags()&Intent.FLAG_RECEIVER_BOOT_UPGRADE) != 0))
154                             == null) {
155                 // Ah, this recipient is unavailable.  Finish it if necessary,
156                 // and mark the broadcast record as ready for the next.
```

```
157                 ......
158                 scheduleBroadcastsLocked();
159                 r.state = BroadcastRecord.IDLE;
160                 return;
161             }
162
163             mPendingBroadcast = r;
164             mPendingBroadcastRecvIndex = recIdx;
165         }
166     }
167
168     ......
169 }
```

从前面的Step 6可以知道，如果广播接收者nextReceiver的类型不是BroadcastFilter，那么它的类型就一定是ResolveInfo，因此，第127行和第128行代码就可以安全地将广播接收者nextReceiver强制转换为一个ResolveInfo对象info。

第131行代码得到ResolveInfo对象info所描述的广播接收者的android:process属性值，即它需要运行在的应用程序进程的名称，并且保存在变量targetProcess中，接着第135行到第145行代码就检查这个应用程序进程是否已经启动起来了。如果这个应用程序进程已经启动起来了，那么第139行就可以直接调用成员函数processCurBroadcastLocked将广播发送给它处理；否则，第149行到第154行的if语句就会先调用成员函数startProcessLocked来启动这个应用程序进程。如果启动失败，那么第158行就会调用成员函数scheduleBroadcastsLocked来结束对BroadcastRecord对象r所描述的广播转发任务的处理；否则，第163行和第164行代码就分别将BroadcastRecord对象r和变量recIdx的值保存在ActivityManagerService类的成员变量mPendingBroadcast和 mPendingBroadcastRecvIndex中，表示ActivityManagerService正在等待BroadcastRecord对象r所描述的广播转发任务的下一个目标广播接收者所在的应用程序进程启动起来。

假设BroadcastRecord对象r所描述的广播转发任务的下一个目标广播接收者是一个动态注册的广播接收者，或者是一个静态注册的，并且是已经启动了的广播接收者，那么这一步执行完成之后，接下来就会调用ActivityManagerService类的成员函数deliverToRegisteredReceiverLocked将一个广播转发给它处理。

Step 10：ActivityManagerService.deliverToRegisteredReceiverLocked

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     private final void deliverToRegisteredReceiverLocked(BroadcastRecord r,
06             BroadcastFilter filter, boolean ordered) {
07         boolean skip = false;
08         if (filter.requiredPermission != null) {
09             int perm = checkComponentPermission(filter.requiredPermission,
10                     r.callingPid, r.callingUid, -1);
11             if (perm != PackageManager.PERMISSION_GRANTED) {
12                 ......
13                 skip = true;
14             }
15
16         }
17         if (r.requiredPermission != null) {
18             int perm = checkComponentPermission(r.requiredPermission,
19                     filter.receiverList.pid, filter.receiverList.uid, -1);
20             if (perm != PackageManager.PERMISSION_GRANTED) {
21                 ......
22                 skip = true;
```

```
23              }
24
25          }
26
27          if (!skip) {
28              ......
29
30              try {
31                  ......
32                  performReceiveLocked(filter.receiverList.app, filter.receiverList.receiver,
33                          new Intent(r.intent), r.resultCode,
34                          r.resultData, r.resultExtras, r.ordered, r.initialSticky);
35                  ......
36              } catch (RemoteException e) {
37                  ......
38              }
39          }
40
41      }
42
43      ......
44  }
```

ActivityManagerService将一个广播转发给一个目标广播接收者之前，有可能需要检查这个广播的发送者和接收者的权限。这个权限检查是双向的，即需要检查一个广播发送者是否有权限向一个目标广播接收者发送一个广播，以及一个目标广播接收者是否有权限接收一个广播发送者发出来的一个广播。

如果BroadcastFilter 对象filter的成员变量requiredPermission的值不等于null，即第8行的if语句为true，那么就表示ActivityManagerService需要检查广播发送者的权限。如果BroadcastRecord对象r的成员变量requiredPermission的值不等于null，即17行的if语句为true，那么就表示ActivityManagerService需要检查广播接收者的权限。这两个权限主要就是调用ActivityManagerService类的成员函数checkComponentPermission来检查对方的PID以及UID是否符合要求。

如果不需要进行前面的权限检查，或者能够通过前面的权限检查，即变量skip的值等于false，那么接下来第32行到第34行代码就调用成员函数performReceiveLocked将BroadcastRecord对象r所描述的广播转发给BroadcastFilter 对象filter所描述的目标广播接收者处理。

Step 11：ActivityManagerService.performReceiveLocked

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02          implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03      ......
04
05      static void performReceiveLocked(ProcessRecord app, IIntentReceiver receiver,
06              Intent intent, int resultCode, String data, Bundle extras,
07              boolean ordered, boolean sticky) throws RemoteException {
08          // Send the intent to the receiver asynchronously using one-way binder calls.
09          if (app != null && app.thread != null) {
10              // If we have an app thread, do the call through that so it is
11              // correctly ordered with other one-way calls.
12              app.thread.scheduleRegisteredReceiver(receiver, intent, resultCode,
13                      data, extras, ordered, sticky);
14          } else {
15              receiver.performReceive(intent, resultCode, data, extras, ordered, sticky);
16          }
17      }
18
19      ......
20  }
```

参数receiver指向了一个实现了IIntentReceiver接口的Binder代理对象，用来描述目标广播接收者；参数app指向一个ProcessRecord对象，用来描述目标广播接收者所运行在的应用程序进程；参数intent用来描述即将要发送给目标广播接收者的一个广播。

从前面9.2小节的内容可以知道，当一个Activity组件或者一个Service组件将一个广播接收者注册到ActivityManagerService中时，它所在的应用程序进程会先将这个广播接收者封装成一个类型为InnerReceiver的Binder本地对象，然后再注册到ActivityManagerService中。因此，当ActivityManagerService要将一个广播转发给一个目标广播接收者处理时，实际上是将这个广播转发给封装了这个目标广播接收者的一个InnerReceiver对象来处理。

如果目标广播接收者需要通过它所运行在的应用程序进程来间接接收一个广播，那么第12行和第13行就会调用运行在该应用程序进程中的一个ApplicationThread对象的Binder代理对象的成员函数scheduleRegisteredReceiver来向它发送这个广播；否则，第15行就直接调用与它关联的一个IntentReceiver对象的Binder代理对象的成员函数performReceive来向它发送这个广播。

ActivityManagerService向一个应用程序进程发送一个广播时，采用的是异步进程间通信方式。在前面的5.1.1小节中分析Binder驱动程序中的结构体binder_node时提到，发送给一个Binder实体对象的所有异步事务都是保存在一个异步事务队列中的。由于保存在一个异步事务队列中的异步事务在同一时刻只有一个会得到处理，即只有位于队列头部的异步事务才会得到处理，因此，ActivityManagerService就可以保证它发送给同一个应用程序进程的所有广播都可以按照发送顺序来串行地被接收和处理。

假设目标广播接收者需要通过它所运行在的应用程序进程来间接接收一个广播，即参数app以及参数app的成员变量thread的值都不等于null，那么接下来ActivityManagerService就会执行第12行和第13行代码，以便可以将参数intent所描述的广播发送给该应用程序进程处理。在我们这个情景中，参数app描述的应用程序进程即为应用程序Broadcounter所运行在的进程，而参数app的成员变量thread即为引用了运行在该应用程序进程中的一个ApplicationThread对象的Binder代理对象，即它是一个ApplicationThreadProxy对象。接下来，我们就继续分析ApplicationThreadProxy类的成员函数scheduleRegisteredReceiver的实现。

Step 12：ApplicationThreadProxy.scheduleRegisteredReceiver

frameworks/base/core/java/android/app/ApplicationThreadNative.java

```
01 class ApplicationThreadProxy implements IApplicationThread {
02     ......
03
04     public void scheduleRegisteredReceiver(IIntentReceiver receiver, Intent intent,
05             int resultCode, String dataStr, Bundle extras, boolean ordered, boolean sticky)
06             throws RemoteException {
07         Parcel data = Parcel.obtain();
08         data.writeInterfaceToken(IApplicationThread.descriptor);
09         data.writeStrongBinder(receiver.asBinder());
10         intent.writeToParcel(data, 0);
11         data.writeInt(resultCode);
12         data.writeString(dataStr);
13         data.writeBundle(extras);
14         data.writeInt(ordered ? 1 : 0);
15         data.writeInt(sticky ? 1 : 0);
16         mRemote.transact(SCHEDULE_REGISTERED_RECEIVER_TRANSACTION, data, null,
17             IBinder.FLAG_ONEWAY);
18         data.recycle();
19     }
20
21     ......
22 }
```

第8行到第15行代码将前面传进来的参数写入到Parcel对象data中，接着再通过ApplicationThreadProxy类内部的一个Binder代理对象mRemote向应用程序Broadcounter发送一个类型为SCHEDULE_REGISTERED_RECEIVER_TRANSACTION的进程间通信请求。

注意

这个进程间通信请求是异步的，因为第16行和第17行代码将它的传输标志位设置为IBinder.FLAG_ONEWAY。

以上8步都是在ActivityManagerService中执行的，接下来第13步到第17步是在应用程序Broadcounter中执行的，主要是用来处理ActivityManagerService发出的类型为SCHEDULE_REGISTERED_RECEIVER_TRANSACTION的进程间通信请求，如图9–5所示。

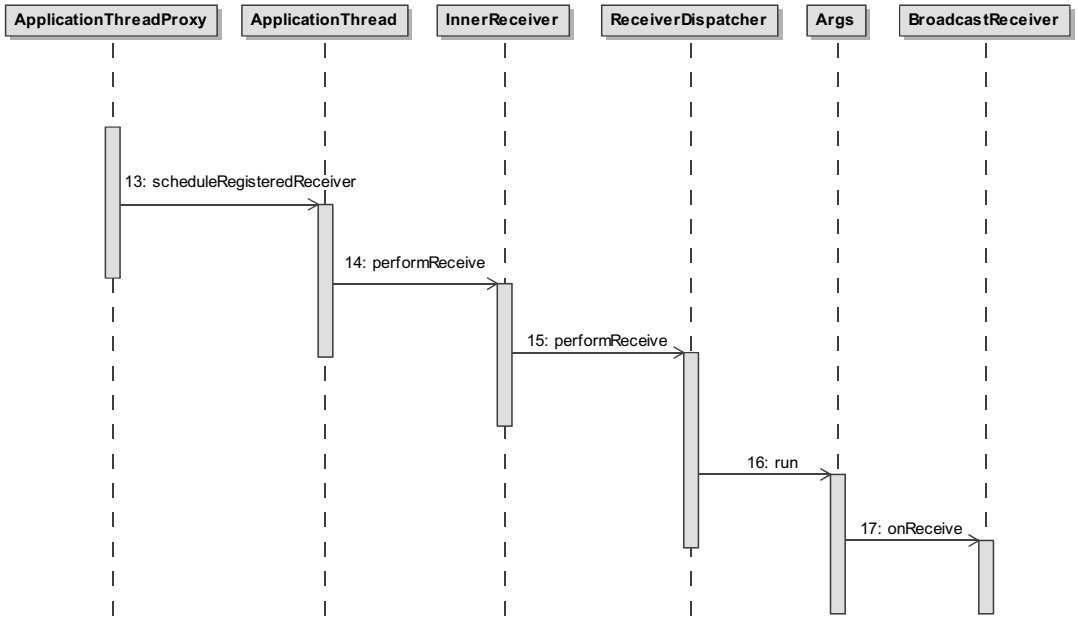


图9–5 广播的发送过程——Step 13至Step 17

Step 13: ApplicaitonThread.scheduleRegisteredReceiver

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final class ApplicationThread extends ApplicationThreadNative {
05         ......
06
07         // This function exists to make sure all receiver dispatching is
08         // correctly ordered, since these are one-way calls and the binder driver
09         // applies transaction ordering per object for such calls.
10         public void scheduleRegisteredReceiver(IIntentReceiver receiver, Intent intent,
11                 int resultCode, String dataStr, Bundle extras, boolean ordered,
12                 boolean sticky) throws RemoteException {
13             receiver.performReceive(intent, resultCode, dataStr, extras, ordered, sticky);
14         }
15
16         ......
17     }
18
19     ......
20 }
```

ApplicationThread类的成员函数scheduleRegisteredReceiver用来处理类型为SCHEDULE_REGISTERED_RECEIVER_TRANSACTION的进程间通信请求。

参数receiver指向了一个InnerReceiver对象。从前面9.2小节中的Step 3可以知道，每一个InnerReceiver对象在内部都封装了一个广播接收者，并且代替它所封装的广播接收者注册到ActivityManagerService中。这样，当ActivityManagerService将一个广播发送给一个目标广播接收者时，实际上是将这个广播发送给了与目标广播接收者相关联的一个InnerReceiver对象，而这个InnerReceiver对象是通过它的成员函数performReceive来接收这个广播的，如第13行代码所示。

Step 14：InnerReceiver.performReceive

frameworks/base/core/java/android/app/LoadedApk.java

```
01 final class LoadedApk {
02     ......
03
04     static final class ReceiverDispatcher {
05
06         final static class InnerReceiver extends IIntentReceiver.Stub {
07             final WeakReference<LoadedApk.ReceiverDispatcher> mDispatcher;
08             ......
09
10             public void performReceive(Intent intent, int resultCode,
11                     String data, Bundle extras, boolean ordered, boolean sticky) {
12                 LoadedApk.ReceiverDispatcher rd = mDispatcher.get();
13                 ......
14                 if (rd != null) {
15                     rd.performReceive(intent, resultCode, data, extras,
16                             ordered, sticky);
17                 }
18                 ......
19             }
20         }
21
22         ......
23     }
24
25     ......
26 }
```

在前面9.2小节中的Step 3提到，InnerReceiver类的成员变量mDispatcher是一个类型为ReceiverDispatcher的弱引用，因此，第15行和第16行代码实际上是调用了ReceiverDispatcher类的成员函数performReceive来接收参数intent所描述的一个广播的。

Step 15：ReceiverDispatcher.performReceive

frameworks/base/core/java/android/app/LoadedApk.java

```
01 final class LoadedApk {
02     ......
03
04     static final class ReceiverDispatcher {
05         ......
06         final Handler mActivityThread;
07         ......
08
09         public void performReceive(Intent intent, int resultCode,
10                 String data, Bundle extras, boolean ordered, boolean sticky) {
11             ......
12
13             Args args = new Args();
```

```
14              args.mCurIntent = intent;
15              args.mCurCode = resultCode;
16              args.mCurData = data;
17              args.mCurMap = extras;
18              args.mCurOrdered = ordered;
19              args.mCurSticky = sticky;
20              if (!mActivityThread.post(args)) {
21                  ......
22              }
23          }
24
25          ......
26      }
27
28      ......
29  }
```

在前面的Step 3中提到，ReceiverDispatcher类的成员变量mActivityThread是一个Handler对象，它指向了ActivityThread类的成员变量mH，是用来向应用程序Broadcounter的主线程的消息队列发送消息的。

第13行到第19行代码首先将参数intent所描述的一个广播封装成一个Args对象，然后将这个Args对象封装成一个消息，最后将这个消息发送到应用程序Broadcounter的主线程的消息队列中。这个消息最终是在Args类的成员函数run中处理的。

Step 16：Args.run

frameworks/base/core/java/android/app/LoadedApk.java

```
01 final class LoadedApk {
02      ......
03
04      static final class ReceiverDispatcher {
05          ......
06          final BroadcastReceiver mReceiver;
07          final boolean mRegistered;
08          ......
09
10          final class Args implements Runnable {
11              private Intent mCurIntent;
12              private boolean mCurOrdered;
13              ......
14
15              public void run() {
16                  BroadcastReceiver receiver = mReceiver;
17                  ......
18
19                  IActivityManager mgr = ActivityManagerNative.getDefault();
20                  Intent intent = mCurIntent;
21                  ......
22
23                  try {
24                      ......
25                      receiver.onReceive(mContext, intent);
26                  } catch (Exception e) {
27                      ......
28                  }
29
30                  if (mRegistered && mCurOrdered) {
31                      try {
32                          ......
33                          mgr.finishReceiver(mIIntentReceiver,
34                                  receiver.getResultCode(),
35                                  receiver.getResultData(),
```

```
36                                  receiver.getResultExtras(false),
37                                  receiver.getAbortBroadcast());
38                      } catch (RemoteException ex) {
39                      }
40                  }
41
42              }
43
44              ......
45          }
46
47          ......
48      }
49
50      ......
51  }
```

ReceiverDispatcher类的成员变量mReceiver指向了一个广播接收者；Args类的成员变量mCurIntent用来描述一个广播，它的目标广播接收者便是ReceiverDispatcher类的成员变量mReceiver所指向的一个广播接收者，因此，第25行就调用这个广播接收者的成员函数onReceive来接收这个广播。

ReceiverDispatcher类的成员变量mRegistered用来描述成员变量mReceiver所指向的广播接收者是否已经注册到ActivityManagerService中。如果已经注册了，那么它的值就等于true；否则，就等于false。此外，Args类的成员变量mCurOrdered用来描述成员变量mCurIntent所描述的广播是否是一个有序广播。如果是，那么它的值就等于true；否则，就等于false。第30行的if语句检查当前正在处理的广播是否是一个有序广播，并且它的目标广播接收者是否已经注册到了ActivityManagerService中。如果两个条件都满足，那么第33行到第37行代码就需要调用ActivityManagerService代理对象mgr的成员函数finishReceiver来通知ActivityManagerService，它前面所转发出来的一个有序广播已经处理完成了。这时候，ActivityManagerService就可以继续将这个有序广播转发给下一个目标广播接收者处理。

在我们这个场景中，ReceiverDispatcher类的成员变量mReceiver所指向的一个广播接收者即为Broadcounter组件在内部实现的一个广播接收者counterActionReceiver，因此，接下来就会调用它的成员函数onReceive来接收一个广播。

Step 17：BroadcastReceiver.onReceive

packages/experimental/Broadcounter/src/shy/luo/broadcounter/Broadcounter.java

```
01 public class Broadcounter extends Activity implements OnClickListener {
02     ......
03
04     private BroadcastReceiver counterActionReceiver = new BroadcastReceiver(){
05         public void onReceive(Context context, Intent intent) {
06             int counter = intent.getIntExtra(CounterService.COUNTER_VALUE, 0);
07             String text = String.valueOf(counter);
08             counterText.setText(text);
09         }
10     }
11
12     ......
13 }
```

当一个广播接收者的成员函数onReceive被调用时，那么就说明它接收到一个使用参数intent来描述的广播了。另外一个参数context用来描述当初将这个广播接收者注册到ActivityManagerService中的Activity组件或者Service组件。

至此，一个广播的发送过程就分析完成了。结合前面的广播机制应用实例，以及广播接收者的注册过程，我们就可以对Android系统的广播机制有一个清晰的认识。

第 10 章

Content Provider组件的实现原理

在前面三章中，我们已经对Android应用程序的三种组件Activity、Service和Broadcast Receiver进行了分析，本章将分析最后一种Android应用程序组件——Content Provider。Content Provider组件除了可以用来作为数据访问接口之外，还可以在不同的应用程序之间进行数据共享。

我们知道，在Android系统中，每一个应用程序被安装时，都会分配到一个不同的Linux用户ID[1]。这样，Android系统就可以基于Linux用户ID来保护每一个应用程序的数据不会被其他应用程序破坏，即每一个应用程序只可以访问自己创建的数据。然而，有时候我们需要在不同的应用程序之间进行数据共享，例如，很多应用程序都需要访问通讯录中的联系人信息来实现自己的业务功能。由于通讯录本身是一个独立的应用程序，因此，其他应用程序是不能直接访问它的联系人信息的，这时候就需要使用Content Provider组件来共享通讯录中的联系人信息了。

由于Content Provider组件可以在不同的应用程序之间进行数据共享，因此，它在软件平台建设中是非常有用的。一个软件平台无非就是由数据和业务组成的，一般来说，数据是统一的，而业务则是百花齐放的。由于不同的业务均使用到了统一的数据，因此，从垂直的方向来看，一个软件平台至少要由数据层、数据访问层和业务层构成；而从水平的方向来看，业务层应该由一系列相对独立的模块组成，以便可以适应越来越复杂的业务环境。在Android系统中，数据层可以使用数据库、文件或者网络来实现，业务层可以使用一系列应用来实现，而数据访问层可以使用Content Provider组件来实现。整个软件平台架构如图10-1所示。

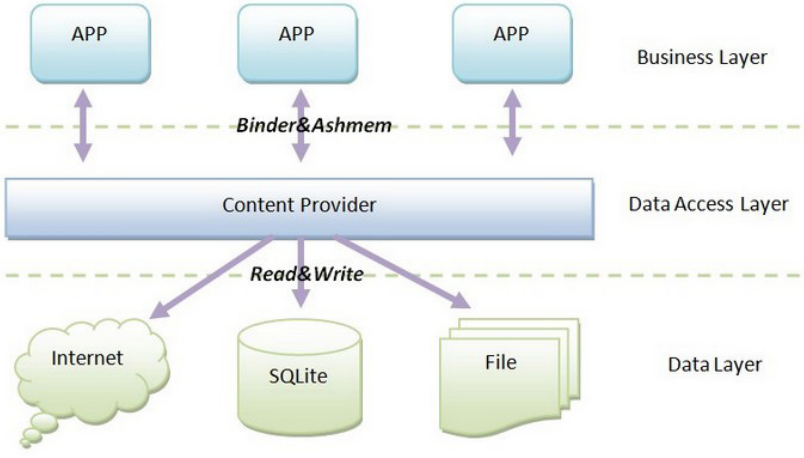


图10-1　基于Android系统的软件平台架构

1　我们也可以配置两个应用程序，使得它们共享同一个Linux用户ID，具体可以参考第16章——Android应用程序的安装和显示过程。

在这个软件平台架构中，为了降低业务层中各个应用之间的耦合度，每一个应用都使用一个Android应用程序来实现，并且它们都是运行在独立的进程中的。同样，为了降低业务层和数据层的耦合度，我们也将数据访问层，即Content Provider组件，运行在一个独立的应用程序进程中。通过这样的划分，Content Provider组件就可以按照自己的方式来管理平台数据，而上层的Android应用程序不需要关心它的具体实现，只要和它约定好数据访问接口就行了。

现在的关键问题是，Content Provider组件是如何将它里面的数据共享给业务层中的Android应用程序访问的呢？我们注意到，Content Provider组件是运行在一个独立的应用程序进程中的，即它本身也是一个Android应用程序，因此，业务层中的Android应用程序是肯定不可以直接访问它的数据的。从前面第5章的内容可以知道，不同的应用程序进程可以通过Binder进程间通信机制来通信，因此，Content Provider组件就可以通过Binder进程间通信机制将它里面的数据传递给业务层中的Android应用程序。然而，事情没有那么简单，因为Content Provider组件一次传递给业务层中的Android应用程序的数据的量可能是非常大的。在这种情况下，如果直接使用Binder进程间通信机制来传递数据，那么数据传输效率就会成为问题。从前面第6章的内容可以知道，不同的应用程序进程可以通过匿名共享内存来传输大数据，因为无论多大的数据，对匿名共享内存来说，需要在进程间传递的仅仅是一个文件描述符而已。这样，结合Binder进程间通信机制以及匿名共享内存机制，Content Provider组件就可以高效地将它里面的数据传递给业务层中的Android应用程序访问了。

下面我们首先开发一个Content Provider组件应用实例，然后结合这个应用实例来分析Content Provider组件的实现原理，包括分析Content Provider组件的启动过程、数据共享原理，以及数据更新通知机制。

10.1　Content Provider组件应用实例

在本节中，我们将开发两个应用程序ArticlesProvider和Article。其中，应用程序ArticlesProvider由一个Content Provider组件ArticlesProvider组成，它在内部使用一个SQLite数据库来维护一个博客文章列表；而应用程序Article由两个Activity组件MainActivity和ArticleActivity组成，它们用来浏览、添加、修改和删除ArticlesProvider组件中的博客文章列表。

接下来，我们就分别介绍应用程序ArticlesProvider和Article的实现。

10.1.1　ArticlesProvider

应用程序ArticlesProvider的目录结构如下：

```
~/Android/packages/experimental/ArticlesProvider
----AndroidManifest.xml
----Android.mk
----src
    ----shy/luo/providers/articles
        ----Articles.java
        ----ArticlesProvider.java
----res
    ----values
        ----strings.xml
    ----drawable
        ----icon.png
```

它包含了一个源代码目录src、一个资源目录res、一个配置文件AndroidManifest.xml和一个编译脚本文件Android.mk。接下来我们分别介绍每一个文件的内容。

Articles.java

```
01 package shy.luo.providers.articles;
02
03 import android.net.Uri;
04
05 public class Articles {
06     /*Data Field*/
07     public static final String ID = "_id";
08     public static final String TITLE = "_title";
09     public static final String ABSTRACT = "_abstract";
10     public static final String URL = "_url";
11
12     /*Default sort order*/
13     public static final String DEFAULT_SORT_ORDER = "_id asc";
14
15     /*Call Method*/
16     public static final String METHOD_GET_ITEM_COUNT = "METHOD_GET_ITEM_COUNT";
17     public static final String KEY_ITEM_COUNT = "KEY_ITEM_COUNT";
18
19     /*Authority*/
20     public static final String AUTHORITY = "shy.luo.providers.articles";
21
22     /*Match Code*/
23     public static final int ITEM = 1;
24     public static final int ITEM_ID = 2;
25     public static final int ITEM_POS = 3;
26
27     /*MIME*/
28     public static final String CONTENT_TYPE = "vnd.android.cursor.dir/vnd.shy.luo.article";
29     public static final String CONTENT_ITEM_TYPE = "vnd.android.cursor.item/vnd.shy.luo.article";
30
31     /*Content URI*/
32     public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/item");
33     public static final Uri CONTENT_POS_URI = Uri.parse("content://" + AUTHORITY + "/pos");
34 }
```

第7行到第10行定义了四个常量ID、TITLE、ABSTRACT和URL，它们用来描述一个SQLite数据库表的结构（Schema），这个数据库表是用来保存博客文章列表的，它的名称为ArticlesTable。接着第13行又定义了一个常量DEFAULT_SORT_ORDER，用来指定数据库表ArticlesTable中的博客文章条目按照ID字段的升序排列。

数据库表ArticlesTable的结构如图10–2所示。

_id	_title	_abstract	_url

图10–2　数据库表ArticlesTable的结构

第一个字段_id用来描述博客文章条目的ID，它的类型为自动递增的整数，同时它也是一个主键；后面三个字段_title、_abstract和_url分别用来描述博客文章条目的标题、摘要和URL，它们的类型均为字符串。

注意

我们一般将数据库表的主键的字段名称设置为“_id”，因为有一些数据库访问接口，例如CursorAdapter，必须要通过这个字段来标志一行数据。

第16行定义了一个常量METHOD_GET_ITEM_COUNT，用来描述ArticlesProvider组件的一个数据访问接口。Content Provider组件有一个未公开的成员函数call，用来作为通用数据访问入口，它的返回值是一个Key-Value集合，用一个Bundle对象来描述。我们在开发ArticlesProvider组件时，实现了这个未公开的成员函数call，并且提供了一个数据访问接口METHOD_GET_ITEM_COUNT，用来获得博客文章条目的总数。这个总数是以第17行所定义的一个常量KEY_ITEM_COUNT为关键字写入到一个Bundle对象中的。

第20行到第33行定义的八个常量AUTHORITY、ITEM、ITEM_ID、ITEM_POS、CONTENT_TYPE、CONTENT_ITEM_TYPE、CONTENT_URI和CONTENT_POS_URI都是用来描述ArticlesProvider组件的访问URI的。

每一个Content Provider组件都是通过一个URI来访问的。URI的全称是Universal Resource Identifier，它的结构（Schema）与我们常见的URL类似，由四个组件构成，如图10-3所示。

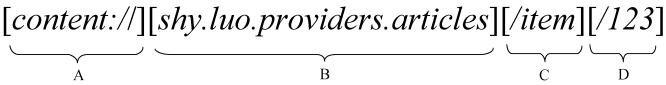


图10-3　Content Provider组件的URI结构

A组件是一个协议名称，它的值固定为“content://”，是Content Provider组件的专用访问协议。

B组件是一个Content Provider组件的android:authority属性值。这个组件类似于URL中的域名，因此，我们需要保证它是全局唯一的，一般就使用它所描述的Content Provider组件的包名来命名。例如，ArticlesProvider组件的包名为“shy.luo.providers.articles”，于是，我们就将它的android:authority属性值设置为“shy.luo.providers.articles”。

C组件是一个资源相对路径，用来描述要访问的资源的类型。如果一个Content Provider组件只有一种资源，那么就可以忽略这个组件；否则，就要通过它来指定要访问的资源的类型。假设一个Content Provider组件有两种电脑数据可以访问，其中一种是笔记本电脑数据，另一种是平板电脑数据，它们对应的资源类型就定义为notebook和pad。如果我们需要获得的是笔记本电脑数据，那么就将资源相对路径设置为“/notebook”；如果我们需要获得的是平板电脑数据，那么就将资源相对路径设置为“/pad”。如果我们需要进一步将笔记本电脑和平板电脑分别细分为装了Windows系统和Linux系统的，以及装了iOS系统和Android系统的，那么就可以在原来的资源相对路径的基础上再增加一个相对路径，如“/notebook/windows”和“/notebook/linux”、“/pad/ios”和“/pad/android”。

D组件是一个资源ID，用来描述要访问的一个具体资源。我们通常使用一个数值来描述一个具体的资源。继续以前面提供电脑数据访问的Content Provider组件为例，假设我们需要获得的是第123号装了Android系统的平板电脑数据，那么就将资源ID指定为123，并且将相对路径指定为“/pad/android”，两者连在一起就得到最终的访问路径为“/pad/android/123”。资源ID是可选的，如果没有指定，那么就表示要访问的是某一种类型资源的所有实例。假设我们需要获得所有装了Android系统的平板电脑数据，那么就将访问路径指定为“/pad/android/”即可。

第32行和第33行定义了COTENT_URI和CONTENT_POS_URI两个URI，它们分别通过ID值和位置值来获得ArticlesProvider组件中的一个博客文章条目。例如，“content://shy.luo.providers.articles/item/123”表示要获得ID值为123的博客文章条目，而“content://shy.luo.providers.articles/pos/1”表示要获得位置值为1的博客文章条目。

每一个URI所访问的资源都具有一种特定的数据类型，这些数据类型使用多功能Internet邮件扩展协议（MIME）来描述。MIME的定义形式为“[type]/[subtype]”，其中，[type]用来描述数据类型的大类，而[subtype]用来描述数据类型的子类。例如，“text/html”描述的数据类型的大类是文本（text），而子类是超文本标记语言（html）。

Content Provider组件规定，它所返回的资源的数据类型统一划分为“vnd.android.cursor.dir”和“vnd.android.cursor.item”两个大类，其中，前者用来描述一个数据集合，后者用来描述一个个体数据。Content Provider组件所返回的资源的数据类型的子类则是可以自定义的，它的定义形式一般约定为“vnd.[company name].[resource type]”，其中，[company name]表示一个公司名称，[resource type]表示一个资源类型。第28行和第29行定义了CONTENT_TYPE和COTENT_ITEM_TYPE两个常量，用来描述ArticlesProvider组件返回的资源的数据类型，其中，前者用来描述一个vnd.shy.luo.article数据集合，即一个博客文章集合；后者用来描述一个vnd.shy.luo.article数据，即一个博客文章条目。

第23行到第25行定义了三个URI匹配码ITEM、ITEM_ID和ITEM_POS，它们用来对ArticlesProvider组件的访问URI进行分类。这些URI可以划分为三类：第一类的形式为“content://shy.luo.providers.articles/item”，对应的URI匹配码为ITEM，表示要获得所有的博客文章；第二类的形式为“content://shy.luo.providers.articles/item/#”，对应的URI匹配码为ITEM_ID，表示要获得ID值为#的博客文章条目；第三类的形式为“content://shy.luo.providers.articles/pos/#”，表示要获得位置值为#的博客文章条目。

接下来，我们就会看到这些常量的具体使用方法。

ArticlesProvider.java

这个文件用来实现ArticlesProvider组件，我们分段来阅读。

```
001 package shy.luo.providers.articles;
002
003 import java.util.HashMap;
004 import android.content.ContentValues;
005 import android.content.Context;
006 import android.content.UriMatcher;
007 import android.content.ContentProvider;
008 import android.content.ContentUris;
009 import android.content.ContentResolver;
010 import android.database.Cursor;
011 import android.database.sqlite.SQLiteDatabase;
012 import android.database.sqlite.SQLiteDatabase.CursorFactory;
013 import android.database.sqlite.SQLiteException;
014 import android.database.sqlite.SQLiteOpenHelper;
015 import android.database.sqlite.SQLiteQueryBuilder;
016 import android.net.Uri;
017 import android.os.Bundle;
018 import android.text.TextUtils;
019 import android.util.Log;
020
021 public class ArticlesProvider extends ContentProvider {
022     private static final String LOG_TAG = "shy.luo.providers.articles.ArticlesProvider";
023
024     private static final String DB_NAME = "Articles.db";
025     private static final String DB_TABLE = "ArticlesTable";
026     private static final int DB_VERSION = 1;
027
028     private static final String DB_CREATE = "create table " + DB_TABLE +
029                             " (" + Articles.ID + " integer primary key autoincrement, " +
030                             Articles.TITLE + " text not null, " +
031                             Articles.ABSTRACT + " text not null, " +
032                             Articles.URL + " text not null);";
```

ArticlesProvider组件在内部使用一个SQLite数据库来管理博客文章信息。

第24行到第26行代码首先定义了三个常量DB_NAME、DB_TABLE和DB_VERSION，其中，DB_NAME和DB_VERSION用来描述这个SQLite数据库的名称和版本号；DB_TABLE用来描述这个SQLite

数据库中的一个数据库表，ArticlesProvider组件内部的博客文章信息就是保存在它里面的。接着第28行到第32行代码又定义了一条SQL语句DB_CREATE，它是用来创建数据库表DB_TABLE的。

```
033     private static final UriMatcher uriMatcher;
034     static {
035         uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
036         uriMatcher.addURI(Articles.AUTHORITY, "item", Articles.ITEM);
037         uriMatcher.addURI(Articles.AUTHORITY, "item/#", Articles.ITEM_ID);
038         uriMatcher.addURI(Articles.AUTHORITY, "pos/#", Articles.ITEM_POS);
039     }
040
041     private static final HashMap<String, String> articleProjectionMap;
042     static {
043         articleProjectionMap = new HashMap<String, String>();
044         articleProjectionMap.put(Articles.ID, Articles.ID);
045         articleProjectionMap.put(Articles.TITLE, Articles.TITLE);
046         articleProjectionMap.put(Articles.ABSTRACT, Articles.ABSTRACT);
047         articleProjectionMap.put(Articles.URL, Articles.URL);
048     }
```

第33行到第39行代码创建和初始化了一个URI匹配器uriMatcher，用来识别一个URI的类别。URI匹配器uriMatcher里面有“content://shy.luo.providers.articles/item”、“content://shy.luo.providers.articles/item/#”和“content://shy.luo.providers.articles/pos/#” 三个URI匹配规则（符号“#”表示一个任意的数字），对应的URI匹配码分别为Articles.ITEM、Articles.ITEM_ID和Articles.ITEM_POS。

第41行到第48行代码创建和初始化了一个字段映射表articleProjectionMap，用来为数据库表ArticlesTable的字段名称设置别名。为数据库表ArticlesTable的字段名称设置了别名之后，ArticlesProvider组件的访问者就不必知道它的真实字段名称，这样就方便了以后扩展和维护。第44行到第47行代码将数据库表ArticlesTable的字段别名和真实名称设置为一样的，在实际应用中，我们可以根据需要来设置不一样的名称。

```
049     private DBHelper dbHelper = null;
050     private ContentResolver resolver = null;
051
052     @Override
053     public boolean onCreate() {
054         Context context = getContext();
055         resolver = context.getContentResolver();
056         dbHelper = new DBHelper(context, DB_NAME, null, DB_VERSION);
057
058         Log.i(LOG_TAG, "Articles Provider Create");
059         return true;
060     }
061
062     @Override
063     public String getType(Uri uri) {
064         switch (uriMatcher.match(uri)) {
065         case Articles.ITEM:
066             return Articles.CONTENT_TYPE;
067         case Articles.ITEM_ID:
068         case Articles.ITEM_POS:
069             return Articles.CONTENT_ITEM_TYPE;
070         default:
071             throw new IllegalArgumentException("Error Uri: " + uri);
072         }
073     }
```

ArticlesProvider组件的成员函数onCreate在ArticlesProvider组件启动时被调用，它主要用来创建一个DBHelper对象和一个ContentResolver对象，并且分别保存在成员变量dbHelper和resolver中。其中，前者

用来辅助打开一个数据库；后者用来通知其他应用程序，ArticlesProvider组件中的数据发生更新了。

ArticlesProvider组件的成员函数getType用来获取一个URI所要访问的资源的数据类型。只有三类URI可以用来访问ArticlesProvider组件，它们分别是“content://shy.luo.providers.articles/item”、“content://shy.luo.providers.articles/item/#”和“content://shy.luo.providers.articles/pos/#”。第一类URI要访问的资源的数据类型为Articles.CONTENT_TYPE，即“vnd.android.cursor.dir/vnd.shy.luo.article”；第二类和第三类URI要访问的资源的数据类型为Articles.CONTENT_ITEM_TYPE，即“vnd.android.cursor.item/vnd.shy.luo.article”。

```
074     @Override
075     public Uri insert(Uri uri, ContentValues values) {
076         if(uriMatcher.match(uri) != Articles.ITEM) {
077             throw new IllegalArgumentException("Error Uri: " + uri);
078         }
079 
080         SQLiteDatabase db = dbHelper.getWritableDatabase();
081 
082         long id = db.insert(DB_TABLE, Articles.ID, values);
083         if(id < 0) {
084             throw new SQLiteException("Unable to insert " + values + " for " + uri);
085         }
086 
087         Uri newUri = ContentUris.withAppendedId(uri, id);
088         resolver.notifyChange(newUri, null);
089 
090         return newUri;
091     }
092 
093     @Override
094     public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {
095         SQLiteDatabase db = dbHelper.getWritableDatabase();
096         int count = 0;
097 
098         switch(uriMatcher.match(uri)) {
099         case Articles.ITEM: {
100             count = db.update(DB_TABLE, values, selection, selectionArgs);
101             break;
102         }
103         case Articles.ITEM_ID: {
104             String id = uri.getPathSegments().get(1);
105             count = db.update(DB_TABLE, values, Articles.ID + "=" + id
106                     + (!TextUtils.isEmpty(selection) ? " and (" + selection + ')' : ""), selectionArgs);
107             break;
108         }
109         default:
110             throw new IllegalArgumentException("Error Uri: " + uri);
111         }
112 
113         resolver.notifyChange(uri, null);
114 
115         return count;
116     }
117 
118     @Override
119     public int delete(Uri uri, String selection, String[] selectionArgs) {
120         SQLiteDatabase db = dbHelper.getWritableDatabase();
121         int count = 0;
122 
123         switch(uriMatcher.match(uri)) {
124         case Articles.ITEM: {
125             count = db.delete(DB_TABLE, selection, selectionArgs);
```

```
126             break;
127         }
128         case Articles.ITEM_ID: {
129             String id = uri.getPathSegments().get(1);
130             count = db.delete(DB_TABLE, Articles.ID + "=" + id
131                     + (!TextUtils.isEmpty(selection) ? " and (" + selection + ')' : ""), selectionArgs);
132             break;
133         }
134         default:
135             throw new IllegalArgumentException("Error Uri: " + uri);
136         }
137 
138         resolver.notifyChange(uri, null);
139 
140         return count;
141     }
```

ArticlesProvider组件的成员函数insert用来向ArticlesProvider组件插入一个新的博客文章条目，其中，参数values用来描述一个要插入的博客文章条目的信息。第80行代码首先通过DBHelper对象dbHelper打开一个SQLite数据库。接着第82行代码将参数values所描述的博客文章条目写入到这个SQLite数据库中。最后第88行代码通过ContentResolver对象resolver来通知其他应用程序，ArticlesProvider组件中的博客文章条目发生改变了。

ArticlesProvider组件的成员函数update用来更新ArticlesProvider组件中的博客文章条目，其中，参数selection和selectionArgs用来指定要更新的博客文章条目；参数values用来描述更新后的博客文章条目。第95行代码首先通过DBHelper对象dbHelper打开一个SQLite数据库。接着第98行的switch语句判断要更新的是一组博客文章条目，还是一个博客文章条目。如果要更新的是一个博客文章条目，那么第104行代码就从参数uri中获得它的ID值。最后第113行代码通过ContentResolver对象resolver来通知其他应用程序，ArticlesProvider组件中的博客文章条目发生改变了。

ArticlesProvider组件的成员函数delete用来删除ArticlesProvider组件中的博客文章条目，其中，参数selection和selectionArgs用来指定要删除的博客文章条目。第120行代码首先通过DBHelper对象dbHelper打开一个SQLite数据库。接着第123行的switch语句判断要删除的是一组博客文章条目，还是一个博客文章条目。如果要删除的是一个博客文章条目，那么第129行代码就从参数uri中获得它的ID值。最后第138行代码通过ContentResolver对象resolver来通知其他应用程序，ArticlesProvider组件中的博客文章条目发生改变了。

```
142     @Override
143     public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) {
144         SQLiteDatabase db = dbHelper.getReadableDatabase();
145 
146         SQLiteQueryBuilder sqlBuilder = new SQLiteQueryBuilder();
147         String limit = null;
148 
149         switch (uriMatcher.match(uri)) {
150         case Articles.ITEM: {
151             sqlBuilder.setTables(DB_TABLE);
152             sqlBuilder.setProjectionMap(articleProjectionMap);
153             break;
154         }
155         case Articles.ITEM_ID: {
156             String id = uri.getPathSegments().get(1);
157             sqlBuilder.setTables(DB_TABLE);
158             sqlBuilder.setProjectionMap(articleProjectionMap);
159             sqlBuilder.appendWhere(Articles.ID + "=" + id);
160             break;
161         }
```

```
162         case Articles.ITEM_POS: {
163             String pos = uri.getPathSegments().get(1);
164             sqlBuilder.setTables(DB_TABLE);
165             sqlBuilder.setProjectionMap(articleProjectionMap);
166             limit = pos + ", 1";
167             break;
168         }
169         default:
170             throw new IllegalArgumentException("Error Uri: " + uri);
171         }
172 
173         Cursor cursor = sqlBuilder.query(db, projection, selection, selectionArgs, null,
null, TextUtils.isEmpty(sortOrder) ? Articles.DEFAULT_SORT_ORDER : sortOrder, limit);
174         cursor.setNotificationUri(resolver, uri);
175 
176         return cursor;
177     }
```

ArticlesProvider组件的成员函数query用来获取ArticlesProvider组件中的博客文章条目，其中，参数selection和selectionArgs用来指定要获取的博客文章条目；参数projection用来指定要获得的博客文章条目的字段的别名。ArticlesProvider组件最终将获取的博客文章条目保存在一个Cursor对象中，并且返回给调用者。

注意

这个Cursor对象中的博客文章条目是按照参数sortOrder所指定的顺序来排列的。

第144行代码首先通过DBHelper对象dbHelper打开一个SQLite数据库。接着第146行代码创建了一个SQLiteQueryBuilder对象sqlBuilder，用来准备一条SQL查询语句，以便可以从前面打开的SQLite数据库中获得调用者指定的博客文章条目。再接着149行的switch语句判断要获取的是一组博客文章条目，还是一个博客文章条目。如果要获取的是一个博客文章条目，那么还需要进一步判断是按照ID值还是位置值来获得这个博客文章条目。如果是按照ID值来获取一个博客文章条目，那么第156行代码就从参数uri中获得它的ID值；否则，第163行代码就从参数uri中获得它的位置值。接下来第173行代码通过SQLiteQueryBuilder对象sqlBuilder来从前面打开的SQLite数据库中获得指定的博客文章条目，并且保存在一个Cursor对象中。在将这个Cursor对象返回给调用者之前，第174行代码会将ContentResolver对象resolver设置到它里面去，以便保存在它里面的博客文章条目发生改变时，它可以通过这个ContentResolver对象resolver来通知其他应用程序，ArticlesProvider组件中的博客文章条目发生改变了。

```
178     @Override
179     public Bundle call(String method, String request, Bundle args) {
180         if(method.equals(Articles.METHOD_GET_ITEM_COUNT)) {
181             return getItemCount();
182         }
183 
184         throw new IllegalArgumentException("Error method call: " + method);
185     }
186 
187     private Bundle getItemCount() {
188         SQLiteDatabase db = dbHelper.getReadableDatabase();
189         Cursor cursor = db.rawQuery("select count(*) from " + DB_TABLE, null);
190 
191         int count = 0;
192         if(cursor.moveToFirst()) {
193             count = cursor.getInt(0);
194         }
195 
196         Bundle bundle = new Bundle();
```

```
197         bundle.putInt(Articles.KEY_ITEM_COUNT, count);
198
199         cursor.close();
200         db.close();
201
202         return bundle;
203     }
```

ArticlesProvider组件的成员函数call是一个未公开的数据查询接口。如果一个应用程序想通过这个未公开的接口来获得一个Content Provider组件中的数据，那么它就必须要在Android源代码环境中进行编译。

一般来说，一个应用程序如果想要获取一个Content Provider组件中的数据，那么只要调用它的标准数据查询接口，即调用它的成员函数query就可以了。既然如此，为什么还要提供另外一个非标准的数据查询接口呢？Content Provider组件的成员函数query和call的最大区别就在于，前者将要获取的数据保存在一个Cursor对象中传输给调用者，而后者将要获取的数据保存在一个Bundle对象中传输给调用者。保存在Cursor对象中的数据是通过匿名共享内存来传输的，而保存在Bundle对象中的数据是直接使用Binder进程间通信机制来传输的。当要传输的数据量很大时，使用匿名共享内存是有好处的；但是，当要传输的数据量比较小时，使用匿名共享内存就得不偿失了，因为匿名共享内存不是免费的午餐，它的创建过程和映射过程都是需要开销的。因此，Content Provider组件就提供了这个未公开的成员函数call来方便调用者快速地获取一些量比较小的数据。

Content Provider组件的成员函数call只提供了一个数据获取操作Articles.METHOD_GET_ITEM_COUNT，它用来获取ArticlesProvider组件中的博客文章条目的总数，这是通过调用成员函数getItemCount来实现的。Content Provider组件的成员函数getItemCount首先将ArticlesProvider组件中的博客文章条目的总数以KEY_ITEM_COUNT为关键字写入到一个Bundle对象中，然后再返回给调用者。

```
204     private static class DBHelper extends SQLiteOpenHelper {
205         public DBHelper(Context context, String name, CursorFactory factory, int version) {
206             super(context, name, factory, version);
207         }
208
209         @Override
210         public void onCreate(SQLiteDatabase db) {
211              db.execSQL(DB_CREATE);
212         }
213
214         @Override
215         public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
216              db.execSQL("DROP TABLE IF EXISTS " + DB_TABLE);
217              onCreate(db);
218         }
219     }
220 }
```

第204行到第219行代码实现了一个用来打开SQLite数据库的辅助类DBHelper，它是从SQLiteOpenHelper类继承下来的。

由于打开数据库涉及IO操作，因此，我们应该避免在ArticlesProvider组件启动时去打开它内部的数据库，即避免在它的成员函数onCreate中执行打开数据库的操作；否则，ArticlesProvider组件就可能会由于启动超时而被系统中止。那么ArticlesProvider组件应该在什么时候执行打开数据库的操作呢？最好的时机莫过于是第一次使用这个数据库的时候了。

SQLiteOpenHelper就是这样的一个类，当它的成员函数getReadableDatabase或者getWritableDatabase第一次被调用时，它才会打开与它关联的一个SQLite数据库。我们一般不会直接使用SQLiteOpenHelper类来辅助打开一个SQLite数据库，而是以它为父类来实现一个子类，并且在子类中重写它的成员函数

onCreate和onUpgrade，这样就可以在要打开的SQLite数据库不存在或者版本过低时，创建一个新的数据库或者对旧版本的数据库进行升级。

strings.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3     <string name="app_name">Articles Storage</string>
4     <string name="provider_name">Articles</string>
5 </resources>
```

这是应用程序ArticlesProvider的字符串资源文件，定义了在应用程序中使用到的各个字符串。

icon.png

这是应用程序ArticlesProvider的图标，可以根据需要来放置不同的图片文件。

AndroidManifest.xml

```
01 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
02         package="shy.luo.providers.articles">
03     <application android:process="shy.luo.process.article"
04         android:label="@string/app_name"
05         android:icon="@drawable/icon">
06         <provider android:name="ArticlesProvider"
07             android:authorities="shy.luo.providers.articles"
08             android:label="@string/provider_name"
09             android:multiprocess="false">
10         </provider>
11     </application>
12 </manifest>
```

这是应用程序ArticlesProvider的配置文件。由于在程序中使用到了Content Provider组件ArticlesProvider，因此，我们就需要在这里对它进行配置。在配置一个Content Provider组件时，最重要的就是要指定它的android:authorities属性，以便其他应用程序可以对它进行访问。Content Provider组件还有另外一个重要的属性android:multiprocess，如果它的值等于true，那么就表示这个Content Provider组件可以在每一个访问它的应用程序进程中创建一个实例；否则，这个Content Provider组件在整个系统中只能存在一个实例，并且这个实例是运行在一个独立的应用程序进程中的，其他应用程序只能通过Binder进程间通信机制来访问它。为了使得ArticlesProvider组件能够运行在一个独立的应用程序进程中，我们将它的android:multiprocess属性设置为false。

Android.mk

```
1 LOCAL_PATH:= $(call my-dir)
2 include $(CLEAR_VARS)
3 LOCAL_MODULE_TAGS := optional
4 LOCAL_SRC_FILES := $(call all-subdir-java-files)
5 LOCAL_PACKAGE_NAME := ArticlesProvider
6 include $(BUILD_PACKAGE)
```

这是应用程序ArticlesProvider的编译脚本文件。

上述文件都准备好之后，我们就可以对应用程序ArticlesProvider进行编译和打包了。

```
USER@MACHINE:~/Android$ mmm ./packages/experimental/ArticlesProvider/
USER@MACHINE:~/Android$ make snod
```

编译成功之后，就可以在out/target/product/gerneric/system/app目录下看到应用程序文件ArticlesProvider.apk

了；打包成功之后，应用程序ArticlesProvider就包含在out/target/product/gerneric目录下的Android系统镜像文件system.img中了。

由于在Articles.java文件中定义的常量除了要在ArticlesProvider组件内部使用之外，还要暴露给要访问ArticlesProvider组件的其他应用程序使用。因此，我们最好把编译后的Articles.java文件打包成一个独立的jar包，以便其他应用程序可以引用定义在它里面的常量来访问ArticlesProvider组件。

首先找到编译后的Articles.java文件，即一个Articles.class文件。在Android源代码环境中，进入到目录out/target/common/obj/APPS/ArticlesProvider_intermediates中，找到classes.jar文件，执行jar命令来提取它里面的Articles.class文件。

```
USER@MACHINE:~/Android/out/target/common/obj/APPS/ArticlesProvider_intermediates$ jar -xvf
classes.jar shy/luo/providers/articles/Articles.class
```

然后执行jar命令对Articles.class文件进行打包。

```
USER@MACHINE:~/Android/out/target/common/obj/APPS/ArticlesProvider_intermediates$ jar -cvf
ArticlesProvider.jar ./shy
```

最后，我们就得到了一个ArticlesProvider.jar文件，它里面包含了定义在Articles.java文件中的常量。这样，其他应用程序就可以通过这个ArticlesProvider.jar文件来引用这些常量，以便可以访问ArticlesProvider组件。

至此，ArticlesProvider组件就开发完成了。接下来，我们再开发另外一个应用程序Article，用来访问ArticlesProvider组件中的博客文章信息。

10.1.2　Article

应用程序Article由两个Activity组件MainActivity和ArticleActivity组成，它们用来浏览、添加、修改和删除ArticlesProvider组件中的博客文章信息。

应用程序Article的目录结构如下：

```
~/Android/packages/experimental/Article
----AndroidManifest.xml
----Android.mk
----src
    ----shy/luo/article
        ----Article.java
        ----ArticlesAdapter.java
        ----MainActivity.java
        ----ArticleActivity.java
----libs
    ----ArticlesProvider.jar
----res
    ----layout
        ----main.xml
        ----article.xml
        ----item.xml
    ----values
        ----strings.xml
    ----drawable
        ----border.xml
        ----icon.png
```

它包含了一个源代码目录src、一个资源目录res、一个配置文件AndroidManifest.xml和一个编译脚本文件Android.mk。接下来我们分别介绍每一个文件的内容。

Article.java

```
01 package shy.luo.article;
02
03 public class Article {
04     private int id;
05     private String title;
06     private String abs;
07     private String url;
08
09     public Article(int id, String title, String abs, String url) {
10         this.id = id;
11         this.title = title;
12         this.abs = abs;
13         this.url = url;
14     }
15
16     public void setId(int id) {
17         this.id = id;
18     }
19
20     public int getId() {
21         return this.id;
22     }
23
24     public void setTitle(String title) {
25         this.title = title;
26     }
27
28     public String getTitle() {
29         return this.title;
30     }
31
32     public void setAbstract(String abs) {
33         this.abs = abs;
34     }
35
36     public String getAbstract() {
37         return this.abs;
38     }
39
40     public void setUrl(String url) {
41         this.url = url;
42     }
43
44     public String getUrl() {
45         return this.url;
46     }
47 }
```

这个文件定义了一个Article类，用来封装一个博客文章条目，它里面包括了四个成员变量id、title、abs和url，分别用来描述一个博客文章的ID、标题、摘要和URL。

ArticlesAdapter.java

这个文件实现了一个ArticlesProvider组件的辅助访问类ArticlesAdapter，我们分段来阅读。

```
001 package shy.luo.article;
002
003 import java.util.LinkedList;
004 import shy.luo.providers.articles.Articles;
005 import android.content.ContentResolver;
```

```
006 import android.content.ContentUris;
007 import android.content.ContentValues;
008 import android.content.Context;
009 import android.content.IContentProvider;
010 import android.database.Cursor;
011 import android.net.Uri;
012 import android.os.Bundle;
013 import android.os.RemoteException;
014 import android.util.Log;
015
016
017 public class ArticlesAdapter {
018     private static final String LOG_TAG = "shy.luo.article.ArticlesAdapter";
019
020     private ContentResolver resolver = null;
021
022     public ArticlesAdapter(Context context) {
023         resolver = context.getContentResolver();
024     }
```

ArticlesAdapter类的构造函数会调用参数context的成员函数getContentResolver来获得一个ContentResolver对象，并且保存在成员变量resolver中。后面ArticlesAdapter类就是通过这个ContentResolver对象来访问ArticlesProvider组件中的博客文章信息的。

```
025     public long insertArticle(Article article) {
026         ContentValues values = new ContentValues();
027         values.put(Articles.TITLE, article.getTitle());
028         values.put(Articles.ABSTRACT, article.getAbstract());
029         values.put(Articles.URL, article.getUrl());
030
031         Uri uri = resolver.insert(Articles.CONTENT_URI, values);
032         String itemId = uri.getPathSegments().get(1);
033
034         return Integer.valueOf(itemId).longValue();
035     }
036
037     public boolean updateArticle(Article article) {
038         Uri uri = ContentUris.withAppendedId(Articles.CONTENT_URI, article.getId());
039
040         ContentValues values = new ContentValues();
041         values.put(Articles.TITLE, article.getTitle());
042         values.put(Articles.ABSTRACT, article.getAbstract());
043         values.put(Articles.URL, article.getUrl());
044
045         int count = resolver.update(uri, values, null, null);
046
047         return count > 0;
048     }
049
050     public boolean removeArticle(int id) {
051         Uri uri = ContentUris.withAppendedId(Articles.CONTENT_URI, id);
052
053         int count = resolver.delete(uri, null, null);
054
055         return count > 0;
056     }
```

ArticlesAdapter类的成员函数insertArticle用来向ArticlesProvider组件插入一个新的博客文章条目，其中，参数article用来描述要插入的博客文章条目。第26行到第29行代码首先将参数article的内容写入到一个ContentValues对象values中。接着第31行代码通过ContentResolver对象resolver将ContentValues对象values发送给ArticlesProvider组件，以便ArticlesProvider组件可以往内部的数据库中插入一个新的博

客文章条目。最后第32行代码就可以得到插入成功后的博客文章条目的ID值。

ArticlesAdapter类的成员函数updateArticle用来更新ArticlesProvider组件中的一个博客文章条目，其中，参数article用来描述要更新的博客文章条目，以及更新后的博客文章条目。第38行代码首先构造一个URI，用来指定要更新的博客文章条目的ID值。接着第40行到第43行代码将参数article的内容写入到一个ContentValues对象values中。最后第45行代码通过ContentResolver对象resolver将ContentValues对象values发送给ArticlesProvider组件，以便ArticlesProvider组件可以更新指定的博客文章条目。

ArticlesAdapter类的成员函数removeArticle用来删除ArticlesProvider组件中的一个博客文章条目，其中，参数article用来描述要删除的博客文章条目。第51行代码首先构造一个URI，用来指定要删除的博客文章条目的ID值。然后第53行代码通过ContentResolver对象resolver来通知ArticlesProvider组件删除指定的博客文章条目。

```
057     public LinkedList<Article> getAllArticles() {
058         LinkedList<Article> articles = new LinkedList<Article>();
059 
060         String[] projection = new String[] {
061                 Articles.ID,
062                 Articles.TITLE,
063                 Articles.ABSTRACT,
064                 Articles.URL
065         };
066 
067         Cursor cursor = resolver.query(Articles.CONTENT_URI, projection, null, null, Articles.DEFAULT_SORT_ORDER);
068         if(cursor.moveToFirst()) {
069             do {
070                 int id = cursor.getInt(0);
071                 String title = cursor.getString(1);
072                 String abs = cursor.getString(2);
073                 String url = cursor.getString(3);
074 
075                 Article article = new Article(id, title, abs, url);
076                 articles.add(article);
077             } while(cursor.moveToNext());
078         }
079 
080         return articles;
081     }
082 
083     public int getArticleCount() {
084         int count = 0;
085 
086         try {
087             IContentProvider provider = resolver.acquireProvider(Articles.CONTENT_URI);
088             Bundle bundle = provider.call(Articles.METHOD_GET_ITEM_COUNT, null, null);
089             count = bundle.getInt(Articles.KEY_ITEM_COUNT, 0);
090         } catch(RemoteException e) {
091             e.printStackTrace();
092         }
093 
094         return count;
095     }
```

ArticlesAdapter类的成员函数getAllArticles用来获取ArticlesProvider组件中所有的博客文章条目。ArticlesProvider组件返回来的博客文章条目保存在一个Cursor对象中，因此，第69行到第77行的while循环就依次遍历这个Cursor对象的内容，并且将里面的每一个博客文章条目都封装成一个Article对象，保存在一个LinkedList列表中。最后第80行代码就将这个LinkedList列表返回给调用者。

ArticlesAdapter类的成员函数getArticleCount用来获取ArticlesProvider组件中所有的博客文章条目的

总数，这是通过调用ArticlesProvider组件的成员函数call来实现的。

注意

ArticlesProvider组件返回来的博客文章条目的总数保存在一个Bundle对象中，并且以Articles.KEY_ITEM_COUNT为关键字。

```
096     public Article getArticleById(int id) {
097         Uri uri = ContentUris.withAppendedId(Articles.CONTENT_URI, id);
098 
099         String[] projection = new String[] {
100                 Articles.ID,
101                 Articles.TITLE,
102                 Articles.ABSTRACT,
103                 Articles.URL
104         };
105 
106         Cursor cursor = resolver.query(uri, projection, null, null, Articles.DEFAULT_SORT_ORDER);
107         if (!cursor.moveToFirst()) {
108             return null;
109         }
110 
111         String title = cursor.getString(1);
112         String abs = cursor.getString(2);
113         String url = cursor.getString(3);
114 
115         return new Article(id, title, abs, url);
116     }
117 
118     public Article getArticleByPos(int pos) {
119         Uri uri = ContentUris.withAppendedId(Articles.CONTENT_POS_URI, pos);
120 
121         String[] projection = new String[] {
122                 Articles.ID,
123                 Articles.TITLE,
124                 Articles.ABSTRACT,
125                 Articles.URL
126         };
127 
128         Cursor cursor = resolver.query(uri, projection, null, null, Articles.DEFAULT_SORT_ORDER);
129         if (!cursor.moveToFirst()) {
130             return null;
131         }
132 
133         int id = cursor.getInt(0);
134         String title = cursor.getString(1);
135         String abs = cursor.getString(2);
136         String url = cursor.getString(3);
137 
138         return new Article(id, title, abs, url);
139     }
140 }
```

ArticlesAdapter类的成员函数getArticleById和getArticleByPos分别根据一个博客文章的ID值和位置值从ArticlesProvider组件获得指定的博客文章条目。

注意

它们是通过不同的URI来访问ArticlesProvider组件的，前者使用的URI为Articles.CONTENT_URI，后者使用的URI为Articles.CONTENT_POS_URI。

MainActivity.java

这个文件用来实现应用程序Article的根Activity组件MainActivity，通过它可以浏览ArticlesProvider组件中的博客文章条目，我们分段来阅读。

```
001 package shy.luo.article;
002
003 import shy.luo.providers.articles.Articles;
004 import android.app.Activity;
005 import android.content.Context;
006 import android.content.Intent;
007 import android.database.ContentObserver;
008 import android.net.Uri;
009 import android.os.Bundle;
010 import android.os.Handler;
011 import android.util.Log;
012 import android.view.LayoutInflater;
013 import android.view.View;
014 import android.view.ViewGroup;
015 import android.widget.AdapterView;
016 import android.widget.BaseAdapter;
017 import android.widget.Button;
018 import android.widget.ListView;
019 import android.widget.TextView;
020
021 public class MainActivity extends Activity implements View.OnClickListener, AdapterView.
OnItemClickListener {
022     private final static String LOG_TAG = "shy.luo.article.MainActivity";
023
024     private final static int ADD_ARTICAL_ACTIVITY = 1;
025     private final static int EDIT_ARTICAL_ACTIVITY = 2;
026
027     private ArticlesAdapter aa = null;
028     private ArticleAdapter adapter = null;
029     private ArticleObserver observer = null;
030
031     private ListView articleList = null;
032     private Button addButton = null;
033
034     @Override
035     public void onCreate(Bundle savedInstanceState) {
036         super.onCreate(savedInstanceState);
037         setContentView(R.layout.main);
038
039         aa = new ArticlesAdapter(this);
040
041         articleList = (ListView)findViewById(R.id.listview_article);
042         adapter = new ArticleAdapter(this);
043         articleList.setAdapter(adapter);
044         articleList.setOnItemClickListener(this);
045
046         observer = new ArticleObserver(new Handler());
047         getContentResolver().registerContentObserver(Articles.CONTENT_URI, true, observer);
048
049         addButton = (Button)findViewById(R.id.button_add);
050         addButton.setOnClickListener(this);
051
052         Log.i(LOG_TAG, "Main Activity Created");
053     }
054
055     @Override
056     public void onDestroy() {
```

```
057         super.onDestroy();
058         getContentResolver().unregisterContentObserver(observer);
059     }
```

MainActivity组件的成员函数onCreate主要做了三件事情：第一件事情是创建一个ArticlesAdapter对象aa，用来辅助访问ArticlesProvider组件中的博客文章信息；第二件事情是创建一个ArticleAdapter对象adapter，用来作为界面上一个博客文章列表控件的数据源；第三件事情是创建和注册一个ArticleObserver对象observer，以便可以用来监控ArticlesProvider组件中的博客文章信息的变化。

MainActivity组件的成员函数onDestroy主要用来注销前面所注册的ArticleObserver对象observer，表示它不再需要监控ArticlesProvider组件中的博客文章信息的变化了。

```
060     @Override
061     public void onClick(View v) {
062         if(v.equals(addButton)) {
063             Intent intent = new Intent(this, ArticleActivity.class);
064             startActivityForResult(intent, ADD_ARTICAL_ACTIVITY);
065         }
066     }
067
068     @Override
069     public void onItemClick(AdapterView<?> parent, View view, int pos, long id) {
070         Intent intent = new Intent(this, ArticleActivity.class);
071
072         Article article = aa.getArticleByPos(pos);
073         intent.putExtra(Articles.ID, article.getId());
074         intent.putExtra(Articles.TITLE, article.getTitle());
075         intent.putExtra(Articles.ABSTRACT, article.getAbstract());
076         intent.putExtra(Articles.URL, article.getUrl());
077
078         startActivityForResult(intent, EDIT_ARTICAL_ACTIVITY);
079     }
080
081     @Override
082     public void onActivityResult(int requestCode,int resultCode, Intent data) {
083         super.onActivityResult(requestCode, resultCode, data);
084
085         switch(requestCode) {
086         case ADD_ARTICAL_ACTIVITY: {
087             if(resultCode == Activity.RESULT_OK) {
088                 String title = data.getStringExtra(Articles.TITLE);
089                 String abs = data.getStringExtra(Articles.ABSTRACT);
090                 String url = data.getStringExtra(Articles.URL);
091
092                 Article article = new Article(-1, title, abs, url);
093                 aa.insertArticle(article);
094             }
095
096             break;
097         }
098         case EDIT_ARTICAL_ACTIVITY: {
099             if(resultCode == Activity.RESULT_OK) {
100                 int action = data.getIntExtra(ArticleActivity.EDIT_ARTICLE_ACTION, -1);
101                 if(action == ArticleActivity.MODIFY_ARTICLE) {
102                     int id = data.getIntExtra(Articles.ID, -1);
103                     String title = data.getStringExtra(Articles.TITLE);
104                     String abs = data.getStringExtra(Articles.ABSTRACT);
105                     String url = data.getStringExtra(Articles.URL);
106
107                     Article article = new Article(id, title, abs, url);
108                     aa.updateArticle(article);
109                 } else if(action == ArticleActivity.DELETE_ARTICLE)     {
```

```
110                     int id = data.getIntExtra(Articles.ID, -1);
111                     aa.removeArticle(id);
112                 }
113             }
114
115             break;
116         }
117         }
118     }
```

MainActivity组件的用户界面上有一个列表控件和一个Add按钮，其中，列表控件用来浏览ArticlesProvider组件中的博客文章条目；Add按钮用来启动另外一个Activity组件ArticleActivity，以便可以新增一个博客文章条目。当用户点击列表控件中的一个博客文章条目时，MainActivity组件也会启动ArticleActivity组件，以便可以更新或者删除该博客文章条目。

无论ArticleActivity组件是用来往ArticlesProvider组件中新增一个博客文章条目，还是用来更新和删除一个已有的博客文章条目，当它执行完成之后，都会将执行结果返回给MainActivity组件的成员函数onActivityResult，以便MainActivity组件可以请求ArticlesProvider组件新增、更新或者删除一个博客文章。这是通过调用ArticlesAdapter对象aa的成员函数insertArticle、updateArticle或者removeArticle来实现的。

```
119     private class ArticleObserver extends ContentObserver {
120         public ArticleObserver(Handler handler) {
121             super(handler);
122         }
123
124         @Override
125         public void onChange (boolean selfChange) {
126             adapter.notifyDataSetChanged();
127         }
128     }
```

ContentObserver类是用来监控一个Content Provider组件的数据变化的。MainActivity组件在内部实现了一个ArticleObserver类，它继承了ContentObserver类，用来监控ArticlesProvider组件中的博客文章信息的变化。当ArticlesProvider组件中的博客文章信息发生变化时，即新增、更新或者删除了一个博客文章信息时，ArticleObserver类的成员函数onChange就会被调用，这时候MainActivity组件就会调用ArticleAdapter对象adapter的成员函数notifyDataSetChanged来更新用户界面上的博客文章列表，以便可以同步ArticlesProvider组件中的博客文章信息。

```
129     private class ArticleAdapter extends BaseAdapter {
130         private LayoutInflater inflater;
131
132         public ArticleAdapter(Context context){
133             inflater = LayoutInflater.from(context);
134         }
135
136         @Override
137         public int getCount() {
138             return aa.getArticleCount();
139         }
140
141         @Override
142         public Object getItem(int pos) {
143             return aa.getArticleByPos(pos);
144         }
145
146         @Override
147         public long getItemId(int pos) {
```

```
148             return aa.getArticleByPos(pos).getId();
149         }
150 
151         @Override
152         public View getView(int position, View convertView, ViewGroup parent) {
153             Article article = (Article)getItem(position);
154 
155             if(convertView == null) {
156                 convertView = inflater.inflate(R.layout.item, null);
157             }
158 
159             TextView titleView = (TextView)convertView.findViewById(R.id.textview_article_title);
160             titleView.setText("Title: " + article.getTitle());
161 
162             TextView abstractView = (TextView)convertView.findViewById(R.id.textview_article_abstract);
163             abstractView.setText("Abstract: " + article.getAbstract());
164 
165             TextView urlView = (TextView)convertView.findViewById(R.id.textview_article_url);
166             urlView.setText("URL: " + article.getUrl());
167 
168             return convertView;
169         }
170     }
171 }
```

BaseAdapter类是用来适配一个ListView控件的，它可以定制该ListView控件的用户界面，以及为该ListView控件提供数据源。MainActivity组件在内部实现了一个ArticleAdapter类，它继承了BaseAdapter类，用来作为用户界面上的博客文章列表控件的数据源，以及定制每一个博客文章条目的显示方式。

当MainActivity组件上的博客文章列表需要更新它的内容时，它首先会调用ArticleAdapter类的成员函数getCount来获得它需要显示的项数，然后再调用ArticleAdapter类的成员函数getView来获得每一项需要显示的数据以及界面；而ArticleAdapter类的成员函数getView又是通过调用成员函数getItem来获得每一项所需要显示的数据的。这些数据都来自于ArticlesProvider组件中的博客文章信息，因此，ArticleAdapter类使用了ArticlesAdapter对象aa来访问ArticlesProvider组件。

ArticleActivity.java

这个文件用来实现应用程序Article的子Activity组件ArticleActivity，通过它可以在ArticlesProvider组件中新增、更新和删除一个博客文章条目，我们分段来阅读。

```
001 package shy.luo.article;
002 
003 import shy.luo.providers.articles.Articles;
004 import android.app.Activity;
005 import android.content.Intent;
006 import android.os.Bundle;
007 import android.util.Log;
008 import android.view.View;
009 import android.widget.Button;
010 import android.widget.EditText;
011 
012 public class ArticleActivity extends Activity implements View.OnClickListener {
013     private final static String LOG_TAG = "shy.luo.article.ArticleActivity";
014 
015     public final static String EDIT_ARTICLE_ACTION = "EDIT_ARTICLE_ACTION";
016     public final static int MODIFY_ARTICLE = 1;
017     public final static int DELETE_ARTICLE = 2;
018 
019     private int articleId = -1;
020 
```

```
021     private EditText titleEdit = null;
022     private EditText abstractEdit = null;
023     private EditText urlEdit = null;
024 
025     private Button addButton = null;
026     private Button modifyButton = null;
027     private Button deleteButton = null;
028     private Button cancelButton = null;
029 
030     @Override
031     public void onCreate(Bundle savedInstanceState) {
032         super.onCreate(savedInstanceState);
033         setContentView(R.layout.article);
034 
035         titleEdit = (EditText)findViewById(R.id.edit_article_title);
036         abstractEdit = (EditText)findViewById(R.id.edit_article_abstract);
037         urlEdit = (EditText)findViewById(R.id.edit_article_url);
038 
039         addButton = (Button)findViewById(R.id.button_add_article);
040         addButton.setOnClickListener(this);
041 
042         modifyButton = (Button)findViewById(R.id.button_modify);
043         modifyButton.setOnClickListener(this);
044 
045         deleteButton = (Button)findViewById(R.id.button_delete);
046         deleteButton.setOnClickListener(this);
047 
048         cancelButton = (Button)findViewById(R.id.button_cancel);
049         cancelButton.setOnClickListener(this);
050 
051         Intent intent = getIntent();
052         articleId = intent.getIntExtra(Articles.ID, -1);
053 
054         if(articleId != -1) {
055             String title = intent.getStringExtra(Articles.TITLE);
056             titleEdit.setText(title);
057 
058             String abs = intent.getStringExtra(Articles.ABSTRACT);
059             abstractEdit.setText(abs);
060 
061             String url = intent.getStringExtra(Articles.URL);
062             urlEdit.setText(url);
063 
064             addButton.setVisibility(View.GONE);
065         } else {
066             modifyButton.setVisibility(View.GONE);
067             deleteButton.setVisibility(View.GONE);
068         }
069 
070         Log.i(LOG_TAG, "Article Activity Created");
071     }
```

ArticleActivity组件在启动时，会检查MainActivity组件是否向它传递了一个博客文章ID。如果传递了，那么就说明MainActivity组件启动ArticleActivity组件的目的是更新或者删除一个博客文章条目；否则，就说明MainActivity组件启动ArticleActivity组件的目的是新增一个博客文章条目。

ArticleActivity组件的用户界面上有Add、Modify、Delete和Cancel四个按钮。当ArticleActivity组件是用来新增一个博客文章条目时，Modify和Delete按钮就会被隐藏起来；而当ArticleActivity组件是用来更新或者删除一个博客文章条目时，Add按钮就会被隐藏起来。

```
072     @Override
073     public void onClick(View v) {
```

```
074         if(v.equals(addButton)) {
075             String title = titleEdit.getText().toString();
076             String abs = abstractEdit.getText().toString();
077             String url = urlEdit.getText().toString();
078 
079             Intent result = new Intent();
080             result.putExtra(Articles.TITLE, title);
081             result.putExtra(Articles.ABSTRACT, abs);
082             result.putExtra(Articles.URL, url);
083 
084             setResult(Activity.RESULT_OK, result);
085             finish();
086         } else if(v.equals(modifyButton)){
087             String title = titleEdit.getText().toString();
088             String abs = abstractEdit.getText().toString();
089             String url = urlEdit.getText().toString();
090 
091             Intent result = new Intent();
092             result.putExtra(Articles.ID, articleId);
093             result.putExtra(Articles.TITLE, title);
094             result.putExtra(Articles.ABSTRACT, abs);
095             result.putExtra(Articles.URL, url);
096             result.putExtra(EDIT_ARTICLE_ACTION, MODIFY_ARTICLE);
097 
098             setResult(Activity.RESULT_OK, result);
099             finish();
100         } else if(v.equals(deleteButton)) {
101             Intent result = new Intent();
102             result.putExtra(Articles.ID, articleId);
103             result.putExtra(EDIT_ARTICLE_ACTION, DELETE_ARTICLE);
104 
105             setResult(Activity.RESULT_OK, result);
106             finish();
107         } else if(v.equals(cancelButton)) {
108             setResult(Activity.RESULT_CANCELED, null);
109             finish();
110         }
111     }
```

当用户点击用户界面上的Add按钮时，ArticleActivity组件就会将用户界面上的博客文章信息，即标题、摘要和URL返回给MainActivity组件，以便MainActivity组件可以请求ArticlesProvider组件新增一个博客文章条目。

当用户点击用户界面上的Modify按钮时，ArticleActivity组件也会将用户界面上的博客文章信息返回给MainActivity组件，以便MainActivity组件可以请求ArticlesProvider组件更新一个博客文章条目。

当用户点击用户界面上的Delete按钮时，ArticleActivity组件就会将要删除的博客文章的ID值返回给MainActivity组件，以便MainActivity组件可以请求ArticlesProvider组件删除一个博客文章条目。

main.xml

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
03     android:orientation="vertical"
04     android:layout_width="fill_parent"
05     android:layout_height="fill_parent"
06     android:gravity="bottom">
07     <ListView android:id="@+id/listview_article"
08          android:layout_width="fill_parent"
09          android:layout_height="wrap_content"
10          android:layout_weight="1"
11          android:background="@drawable/border"
12          android:choiceMode="singleChoice">
```

```
    </ListView>
    <LinearLayout android:orientation="horizontal"
         android:layout_height="wrap_content"
         android:layout_width="match_parent"
         android:gravity="center"
         android:layout_marginTop="10dp">
         <Button android:id="@+id/button_add"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:paddingLeft="15dp"
              android:paddingRight="15dp"
              android:text="@string/add">
         </Button>
    </LinearLayout>
</LinearLayout>
```

这是MainActivity组件的界面配置文件，它在屏幕中间显示一个ListView控件和一个Add按钮，其中，ListView控件用来显示博客文章列表；Add按钮用来新增博客文章条目。

article.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:gravity="center">
    <LinearLayout android:orientation="horizontal"
        android:layout_height="wrap_content"
        android:layout_width="fill_parent">
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginRight="24dp"
            android:text="@string/title">
        </TextView>
        <EditText android:id="@+id/edit_article_title"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content">
        </EditText>
    </LinearLayout>
    <LinearLayout android:orientation="horizontal"
        android:layout_height="wrap_content"
        android:layout_width="fill_parent">
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/abs">
        </TextView>
        <EditText android:id="@+id/edit_article_abstract"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content" >
        </EditText>
    </LinearLayout>
    <LinearLayout android:orientation="horizontal"
        android:layout_height="wrap_content"
        android:layout_width="fill_parent">
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginRight="27dp"
            android:text="@string/url">
        </TextView>
        <EditText android:id="@+id/edit_article_url"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content" >
```

```
        </EditText>
    </LinearLayout>
    <LinearLayout android:orientation="horizontal"
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:gravity="center"
        android:layout_marginTop="10dp">
        <Button android:id="@+id/button_modify"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/modify">
        </Button>
        <Button android:id="@+id/button_delete"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/delete">
        </Button>
        <Button android:id="@+id/button_add_article"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:paddingLeft="16dp"
            android:paddingRight="16dp"
            android:text="@string/add">
        </Button>
        <Button android:id="@+id/button_cancel"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/cancel">
        </Button>
    </LinearLayout>
</LinearLayout>
```

这是ArticleActivity组件的界面配置文件，它在屏幕中间显示三个EditText控件，以及Add、Modify、Delete和Cancel四个按钮，其中，EditText控件用来显示或者编辑博客文章信息；按钮用来新增、修改和删除博客文章条目。

item.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <TextView android:id="@+id/textview_article_title"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content">
    </TextView>
    <TextView android:id="@+id/textview_article_abstract"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content">
    </TextView>
    <TextView android:id="@+id/textview_article_url"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="10dp">
    </TextView>
</LinearLayout>
```

这是博客文章列表项的界面配置文件，它分别用三个TextView控件来显示一个博客文章的题目、摘要和URL。

border.xml

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <shape xmlns:android="http://schemas.android.com/apk/res/android"
03     android:shape="rectangle">
04     <solid android:color="#ff0000ff"/>
05     <stroke android:width="1dp"
06         android:color="#000000">
07     </stroke>
08     <padding android:left="7dp"
09         android:top="7dp"
10         android:right="7dp"
11         android:bottom="7dp">
12     </padding>
13     <corners android:radius="10dp"/>
14 </shape>
```

这是博客文章列表的背景和边框配置文件。

strings.xml

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <resources>
03     <string name="app_name">Article</string>
04     <string name="article">Article</string>
05     <string name="add">Add</string>
06     <string name="modify">Modify</string>
07     <string name="delete">Delete</string>
08     <string name="title">Title:</string>
09     <string name="abs">Abstract:</string>
10     <string name="url">URL:</string>
11     <string name="ok">OK</string>
12     <string name="cancel">Cancel</string>
13 </resources>
```

这是应用程序Article的字符串资源文件，定义了在应用程序中使用到的各个字符串。

icon.png

这是应用程序Article的图标，可以根据需要来放置不同的图片文件。

ArticlesProvider.jar

这是一个jar库文件，它包含了访问ArticlesProvider组件所需要的常量。

AndroidManifest.xml

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
03     package="shy.luo.article"
04     android:versionCode="1"
05     android:versionName="1.0">
06     <application android:icon="@drawable/icon" android:label="@string/app_name">
07         <activity android:name=".MainActivity"
08             android:label="@string/app_name">
09             <intent-filter>
10                 <action android:name="android.intent.action.MAIN" />
11                 <category android:name="android.intent.category.LAUNCHER" />
12             </intent-filter>
13         </activity>
14         <activity android:name=".ArticleActivity"
15             android:label="@string/article">
```

```
        </activity>
    </application>
</manifest>
```

这是应用程序Article的配置文件。由于在程序中使用到了Activity组件MainActivity和ArticleActivity，因此，我们就需要在这里对它们进行配置。

Android.mk

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_STATIC_JAVA_LIBRARIES := libArticlesProvider
LOCAL_SRC_FILES := $(call all-subdir-java-files)
LOCAL_PACKAGE_NAME := Article
include $(BUILD_PACKAGE)
##################################################
include $(CLEAR_VARS)
LOCAL_PREBUILT_STATIC_JAVA_LIBRARIES := libArticlesProvider:./libs/ArticlesProvider.jar
include $(BUILD_MULTI_PREBUILT)
```

这是应用程序Article的编译脚本文件。

上述文件都准备好之后，我们就可以对应用程序Article进行编译和打包了。

```
USER@MACHINE:~/Android$ mmm ./packages/experimental/Article/
USER@MACHINE:~/Android$ make snod
```

编译成功之后，就可以在out/target/product/gerneric/system/app目录下看到应用程序文件Article.apk了；打包成功之后，应用程序Article就包含在out/target/product/gerneric目录下的Android系统镜像文件system.img中了。

最后，我们使用新得到的系统镜像文件system.img来启动Android模拟器。

```
USER@MACHINE:~/Android$ emulator
```

Android模拟器运行起来之后，我们就可以在应用程序启动器中启动应用程序Article了，即将它的根Activity组件MainActivity启动起来，它的界面如图10–4所示。

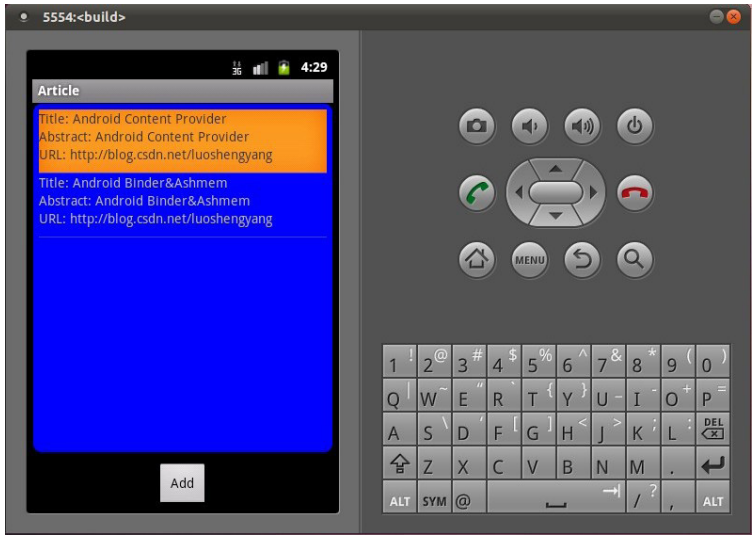


图10–4　MainActivity的用户界面

点击MainActivity组件上的Add按钮，就可以启动ArticleActivity组件来新增一个博客文章条目，如图10–5所示。

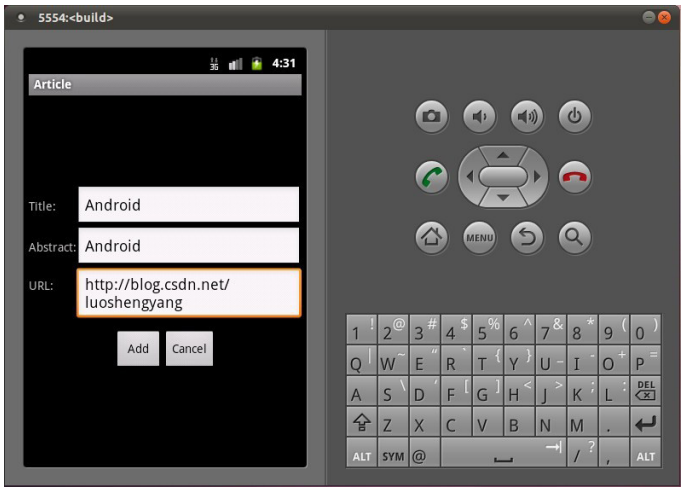


图10–5　在ArticleActivity组件中新增一个博客文章条目

点击MainActivity组件上的博客文章列表项，就可以启动ArticleActivity组件来更新或者删除一个博客文章条目，如图10–6所示。

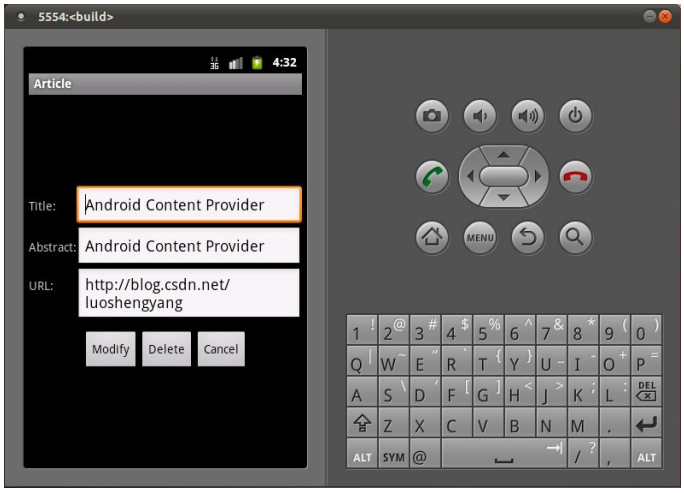


图10–6　在ArticleActivity组件中更新或者删除一个博客文章条目

至此，应用程序Article就介绍完成了。结合前面的应用程序ArticlesProvider，我们就可以对Content Provider组件有一个初步的认识。接下来，我们将结合这两个应用程序来分析Content Provider组件的启动过程、数据共享原理，以及数据更新通知机制。

10.2　Content Provider组件的启动过程

在前面的10.1小节中，我们在应用程序ArticlesProvider中实现了一个Content Provider组件ArticlesProvider，它用来管理一个博客文章列表。当应用程序Article中的MainActivity组件第一次访问ArticlesProvider组件时，系统就会将ArticlesProvider组件启动起来。由于ArticlesProvider组件的android:multiprocess属性被设置为false，因此，系统就会在一个独立的应用程序进程中启动它。本节将以MainActivity组件第一次访问ArticlesProvider组件的过程为例，来说明Content Provider组件在一个独立的应用程序进程中启动的过程。

MainActivity组件第一次访问ArticlesProvider组件的过程如下所示。

（1）MainActivity组件通过Articles.CONTENT_URI来访问ArticlesProvider组件，以便可以获得它的博客文章条目的总数。

（2）MainActivity组件所运行在的应用程序进程发现它里面不存在一个用来访问ArticlesProvider组件的代理对象，于是，它就会通过Articles.CONTENT_URI来请求ActivityManagerService返回一个用来访问ArticlesProvider组件的代理对象。

（3）ActivityManagerService发现ArticlesProvider组件还没有启动起来，于是，它就会先创建一个新的应用程序进程，然后在这个新创建的应用程序进程中启动ArticlesProvider组件。

（4）ArticlesProvider组件启动起来之后，就会将自己发布到ActivityManagerService中，以便ActivityManagerService可以将它的一个代理对象返回给MainActivity组件使用。

这个过程一共分为26个步骤，接下来我们就详细分析每一个步骤。

第1步到第6步如图10-7所示。

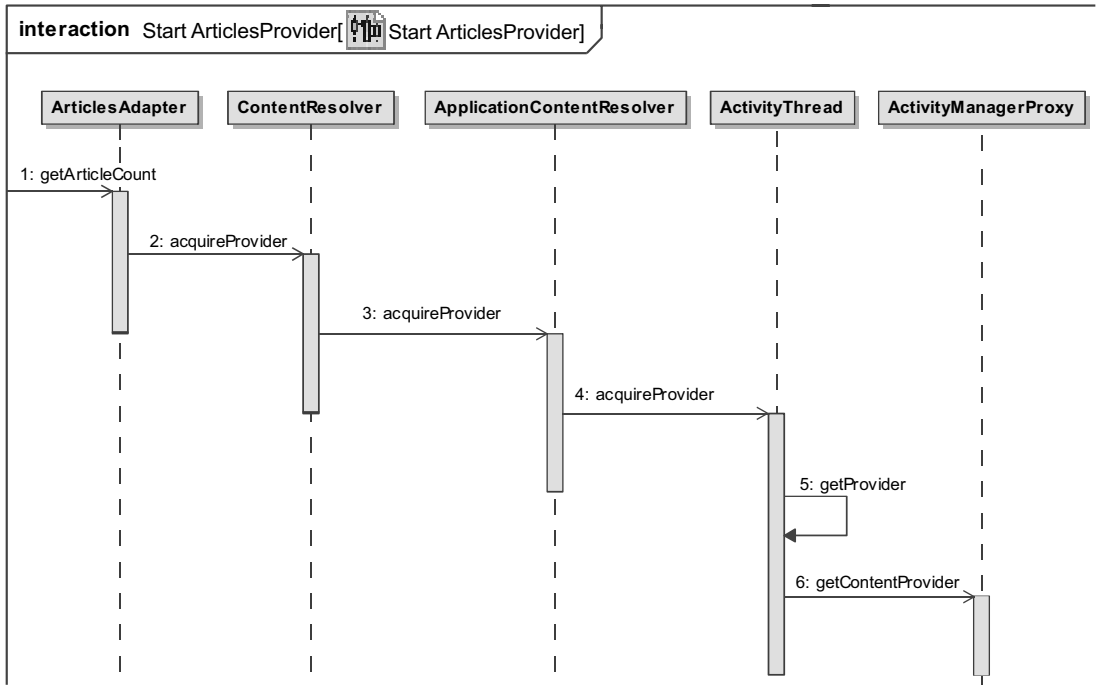


图10-7　ArticlesProvider组件的启动过程——Step 1至Step 6

Step 1：ArticlesAdapter.getArticleCount

packages/experimental/Article/src/shy/luo/article/ArticlesAdapter.java

```
01 public class ArticlesAdapter {
02     ......
03
04     private ContentResolver resolver = null;
05
06     public ArticlesAdapter(Context context) {
07         resolver = context.getContentResolver();
08     }
09     ......
10
11     public int getArticleCount() {
12         int count = 0;
13
14         try {
15             IContentProvider provider = resolver.acquireProvider(Articles.CONTENT_URI);
16             Bundle bundle = provider.call(Articles.METHOD_GET_ITEM_COUNT, null, null);
```

```
17              count = bundle.getInt(Articles.KEY_ITEM_COUNT, 0);
18          } catch(RemoteException e) {
19              e.printStackTrace();
20          }
21
22          return count;
23      }
24
25      ......
26  }
```

MainActivity组件在显示用户界面上的博客文章列表时，会首先调用ArticlesAdapter类的成员函数getArticleCount来获得ArticlesProvider组件中的博客文章条目的总数，这就是MainActivity组件第一次访问ArticlesProvider组件的地方。

ArticlesAdapter类的成员函数getArticleCount使用Articles.CONTENT_URI来访问ArticlesProvider组件，以便可以获得它的博客文章条目的总数。在访问ArticlesProvider组件之前，ArticlesAdapter类首先需要获得它的一个代理对象，这是通过调用ContentResolver对象resolver的成员函数acquireProvider来实现的。

ContentResolver对象resolver是在ArticlesAdapter类的构造函数中获得的，即调用它的参数context的成员函数getContentResolver来获得的。在我们这个场景中，参数context指向了要访问ArticlesProvider组件的MainActivity组件。

MainActivity组件继承了Activity类，而Activity类又继承了ContextWrapper类，并且继承了ContextWrapper类的成员函数getContentResolver。因此，ArticlesAdapter类实际上是通过ContextWrapper类的成员函数getContentResolver来获得ContentResolver对象resolver的。

ContextWrapper类的成员函数getContentResolver的实现如下所示。

frameworks/base/core/java/android/content/ContextWrapper.java

```
01 public class ContextWrapper extends Context {
02     Context mBase;
03     ......
04
05     @Override
06     public ContentResolver getContentResolver() {
07         return mBase.getContentResolver();
08     }
09
10     ......
11 }
```

从前面8.2小节中的Step 2可以知道，ContextWrapper类的成员变量mBase指向了一个ContextImpl对象，它是在MainActivity组件启动时创建的。因此，第7行实际上是调用了ContextImpl类的成员函数getContentResolver来获得一个ContentResolver对象的。

ContextImpl类的成员函数getContentResolver的实现如下所示。

frameworks/base/core/java/android/app/ContextImpl.java

```
01 class ContextImpl extends Context {
02     ......
03
04     private ApplicationContentResolver mContentResolver;
05     ......
06
07     final void init(LoadedApk packageInfo,
08             IBinder activityToken, ActivityThread mainThread,
09             Resources container) {
10         ......
```

```
11
12          mContentResolver = new ApplicationContentResolver(this, mainThread);
13
14          ......
15      }
16      ......
17
18      @Override
19      public ContentResolver getContentResolver() {
20          return mContentResolver;
21      }
22
23      ......
24  }
```

ContextImpl类的成员变量mContentResolver指向了一个ApplicationContentResolver对象，这个ApplicationContentResolver对象是在ContextImpl类的成员函数init中创建的，而ContextImpl类的成员函数init是在启动MainActivity组件时调用的。ContextImpl类的成员函数getContentResolver只是简单地将这个ApplicationContentResolver对象返回给调用者。

回到ArticlesAdapter类的成员函数getArticleCount中，获得了一个ApplicationContentResolver对象之后，接着就调用这个ApplicationContentResolver对象的成员函数acquireProvider来获得ArticlesProvider组件的一个代理对象，即获得与Articles.CONTENT_URI对应的一个Content Provider组件代理对象。

Step 2：ContentResolver.acquireProvider

frameworks/base/core/java/android/content/ContentResolver.java

```
01 public abstract class ContentResolver {
02      ......
03
04      public final IContentProvider acquireProvider(Uri uri) {
05          if (!SCHEME_CONTENT.equals(uri.getScheme())) {
06              return null;
07          }
08          String auth = uri.getAuthority();
09          if (auth != null) {
10              return acquireProvider(mContext, uri.getAuthority());
11          }
12          return null;
13      }
14
15      ......
16  }
```

参数uri是一个URI对象，它的值等于Articles.CONTENT_URI。从前面10.1.1小节的内容可以知道，Articles.CONTENT_URI的值等于“content://shy.luo.providers.articles/item”。

第5行的if语句检查参数uri所描述的URI是否以“content://”开头。如果不是，那么就说明这个URI不是用来访问Content Provider组件的，因此，第6行就直接返回了。

Articles.CONTENT_URI的Authority组件为“shy.luo.providers.articles”，第10行代码以这个字符串为参数调用另外一个重载版本的成员函数acquireProvider来获得ArticlesProvider组件的一个代理对象。

在ContentResolver类中，两个参数版本的成员函数acquireProvider是由其子类实现的。在我们这个场景中，这个子类即为ApplicationContentResolver，因此，接下来就会调用ApplicationContentResolver类的成员函数acquireProvider来获得ArticlesProvider组件的一个代理对象。

Step 3：ApplicationContentResolver.acquireProvider

frameworks/base/core/java/android/app/ContextImpl.java

```
01 class ContextImpl extends Context {
02     ......
03
04     private static final class ApplicationContentResolver extends ContentResolver {
05         public ApplicationContentResolver(Context context, ActivityThread mainThread) {
06             super(context);
07             mMainThread = mainThread;
08         }
09
10         @Override
11         protected IContentProvider acquireProvider(Context context, String name) {
12             return mMainThread.acquireProvider(context, name);
13         }
14         ......
15
16         private final ActivityThread mMainThread;
17     }
18
19     ......
20 }
```

ApplicationContentResolver类的成员变量mMainThread指向了一个ActivityThread对象，它是在构造函数中初始化的，因此，第12行代码实际上是调用了ActivityThread类的成员函数acquireProvider来获得ArticlesProvider组件的一个代理对象，即获得一个名称为“shy.luo.providers.articles”的Content Provider组件代理对象。

Step 4：ActivityThread.acquireProvider

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     public final IContentProvider acquireProvider(Context c, String name) {
05         IContentProvider provider = getProvider(c, name);
06         if(provider == null)
07             return null;
08         ......
09         return provider;
10     }
11
12     ......
13 }
```

第5行代码调用成员函数getProvider来获得一个名称为“shy.luo.providers.articles”的Content Provider组件代理对象。

Step 5：ActivityThread.getProvider

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final IContentProvider getExistingProvider(Context context, String name) {
05         synchronized(mProviderMap) {
06             final ProviderClientRecord pr = mProviderMap.get(name);
07             if (pr != null) {
08                 return pr.mProvider;
09             }
```

```
10              return null;
11          }
12      }
13      ......
14
15      private final IContentProvider getProvider(Context context, String name) {
16          IContentProvider existing = getExistingProvider(context, name);
17          if (existing != null) {
18              return existing;
19          }
20
21          IActivityManager.ContentProviderHolder holder = null;
22          try {
23              holder = ActivityManagerNative.getDefault().getContentProvider(
24                  getApplicationThread(), name);
25          } catch (RemoteException ex) {
26          }
27
28          IContentProvider prov = installProvider(context, holder.provider,
29              holder.info, true);
30          ......
31
32          return prov;
33      }
34
35      ......
36  }
```

ActivityThread类有一个成员变量mProviderMap，它是一个HashMap，用来保存在当前应用程序进程中访问过的Content Provider组件代理对象。第16行代码首先调用成员函数getExistingProvider在这个HashMap中检查是否存在一个与参数name对应的Content Provider组件代理对象。如果存在，那么第18行代码就直接将该Content Provider组件代理对象返回给调用者了；否则，第23行和第24行代码就会首先获得ActivityManagerService的一个代理对象，接着再调用这个代理对象的成员函数getContentProvider来请求ActivityManagerService返回与参数name对应的Content Provider组件代理对象。

ActivityManagerService返回来的Content Provider组件代理对象使用一个ContentProviderHolder对象来描述，接着第28行和第29行代码再调用成员函数installProvider将这个ContentProviderHolder对象封装成一个ProviderClientRecord对象，并且以参数name为关键字保存在成员变量mProviderMap中，这样ActivityThread类以后就不用再请求ActivityManagerService返回与参数name对应的Content Provider组件代理对象了。

接下来，我们首先分析ActivityManagerService代理对象的成员函数getContentProvider的实现，然后再分析ActivityThread类的成员函数installProvider的实现。

Step 6：ActivityManagerProxy.getContentProvider

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
01 class ActivityManagerProxy implements IActivityManager
02 {
03     ......
04
05     public ContentProviderHolder getContentProvider(IApplicationThread caller,
06                                                     String name) throws RemoteException
07     {
08         Parcel data = Parcel.obtain();
09         Parcel reply = Parcel.obtain();
10         data.writeInterfaceToken(IActivityManager.descriptor);
11         data.writeStrongBinder(caller != null ? caller.asBinder() : null);
12         data.writeString(name);
```

```
13          mRemote.transact(GET_CONTENT_PROVIDER_TRANSACTION, data, reply, 0);
14          reply.readException();
15          int res = reply.readInt();
16          ContentProviderHolder cph = null;
17          if (res != 0) {
18              cph = ContentProviderHolder.CREATOR.createFromParcel(reply);
19          }
20          data.recycle();
21          reply.recycle();
22          return cph;
23      }
24
25      ......
26  }
```

第10行到第12行代码将前面传进来的参数写入到Parcel对象data中，然后再通过ActivityManagerProxy类内部的一个Binder代理对象mRemote向ActivityManagerService发送一个类型为GET_CONTENT_PROVIDER_TRANSACTION的进程间通信请求，最后再将ActivityManagerService返回的数据封装成一个ContentProviderHolder对象返回给调用者。

以上6步都是在应用程序Article中执行的，接下来第7步到第9步是在ActivityManagerService中执行的，主要是用来处理 MainActivity组件发出的类型为GET_CONTENT_PROVIDER_TRANSACTION的进程间通信请求，如图10-8所示。

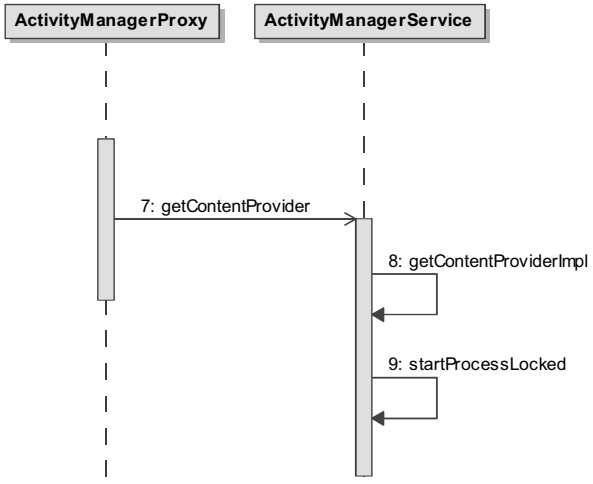


图10-8　ArticlesProvider组件的启动过程——Step 7至Step 9

Step 7：ActivityManagerService.getContentProvider

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02          implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03      ......
04
05      public final ContentProviderHolder getContentProvider(
06              IApplicationThread caller, String name) {
07          ......
08
09          return getContentProviderImpl(caller, name);
10      }
11
12      ......
13  }
```

ActivityManagerService类的成员函数getContentProvider用来处理类型为GET_CONTENT_PROVIDER_

TRANSACTION的进程间通信请求。

第9行调用ActivityManagerService类的另外一个成员函数getContentProviderImpl来进一步获得与参数name对应的一个ContentProviderHolder对象。

Step 8：ActivityManagerService.getContentProviderImpl

这个函数用来获取一个ContentProviderHolder对象，这个ContentProviderHolder对象用来描述一个Content Provider组件的代理对象，我们分段来阅读。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
001 public final class ActivityManagerService extends ActivityManagerNative
002         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
003     ......
004
005     private final ContentProviderHolder getContentProviderImpl(
006             IApplicationThread caller, String name) {
007         ContentProviderRecord cpr;
008         ProviderInfo cpi = null;
009
010         synchronized(this) {
011             ProcessRecord r = null;
012             if (caller != null) {
013                 r = getRecordForAppLocked(caller);
014                 ......
015             }
```

第13行调用成员函数getRecordForAppLocked来获得请求ActivityManagerService返回与参数name对应的Content Provider组件代理对象的应用程序进程的信息，这些信息使用一个ProcessRecord对象r来描述。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
016             // First check if this content provider has been published...
017             cpr = mProvidersByName.get(name);
018             if (cpr != null) {
019                 ......
020             } else {
021                 try {
022                     cpi = AppGlobals.getPackageManager().
023                         resolveContentProvider(name,
024                         STOCK_PM_FLAGS | PackageManager.GET_URI_PERMISSION_PATTERNS);
025                 } catch (RemoteException ex) {
026                 }
027                 ......
028
029                 cpr = mProvidersByClass.get(cpi.name);
030                 final boolean firstClass = cpr == null;
031                 if (firstClass) {
032                     try {
033                         ApplicationInfo ai =
034                             AppGlobals.getPackageManager().
035                             getApplicationInfo(
036                             cpi.applicationInfo.packageName,
037                             STOCK_PM_FLAGS);
038                         ......
039                         cpr = new ContentProviderRecord(cpi, ai);
040                     } catch (RemoteException ex) {
041                         // pm is in same process, this will never happen.
042                     }
043                 }
```

在ActivityManagerService中，每一个已经启动的Content Provider组件都使用一个ContentProviderRecord对象来描述。这些ContentProviderRecord对象分别保存在ActivityManagerService类的两个成员变量

mProvidersByName和mProvidersByClass中。成员变量mProvidersByName所保存的ContentProviderRecord对象是以它们所对应的Content Provider组件的android:authorities属性值为关键字的，而成员变量mProvidersByClass所保存的ContentProviderRecord对象是以它们所对应的Content Provider组件的类名为关键字的。

从前面的调用过程可以知道，参数name指向了ArticlesProvider组件的一个android:authorities属性，它的值等于“shy.luo.providers.articles”。第17行代码首先在成员变量mProvidersByName中检查是否存在与参数name对应的一个ContentProviderRecord对象。如果不存在，即第18行的if语句为false，那么第22行到第24行代码就到PackageManagerService中去查找android:authorities属性值等于name的Content Provider组件的信息，这些信息使用一个ProviderInfo对象cpi来描述。

ProviderInfo对象cpi的成员变量name保存了它所描述的Content Provider组件的类名，第29行代码接着在成员变量mProvidersByClass中检查是否存在与这个类名对应的一个ContentProviderRecord对象。如果不存在，即第31行的if语句为true，那么就说明android:authorities属性值等于name的Content Provider组件还没有启动起来。接下来第33行到第37行代码就到PackageManagerService中去查找它所属的应用程序的信息，这些信息使用一个ApplicationInfo对象ai来描述。最后第39行代码将ProviderInfo对象cpi和ApplicationInfo对象ai封装成一个ContentProviderRecord对象cpr，用来描述即将要启动的Content Provider组件。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
044                 if (r != null && cpr.canRunHere(r)) {
045                     // If this is a multiprocess provider, then just return its
046                     // info and allow the caller to instantiate it.  Only do
047                     // this if the provider is the same user as the caller's
048                     // process, or can run as root (so can be in any process).
049                     return cpr;
050                 }
```

在启动ContentProviderRecord对象cpr所描述的Content Provider组件之前，第44行的if语句首先检查这个Content Provider组件的android: multiprocess属性值是否等于true。如果等于true，那么就说明这个Content Provider组件需要在访问它的应用程序进程中启动。因此，第49行代码就直接将ContentProviderRecord对象cpr返回给需要访问它的应用程序进程，以便这个应用程序进程可以根据保存在它里面的信息来启动需要访问的Content Provider组件。

在我们这个场景中，ContentProviderRecord对象cpr所描述的Content Provider组件即为ArticlesProvider组件，它的android:multiprocess属性值被设置为false，因此，ActivityManagerService就需要在一个独立的应用程序进程中启动它。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
051                 // This is single process, and our app is now connecting to it.
052                 // See if we are already in the process of launching this
053                 // provider.
054                 final int N = mLaunchingProviders.size();
055                 int i;
056                 for (i=0; i<N; i++) {
057                     if (mLaunchingProviders.get(i) == cpr) {
058                         break;
059                     }
060                 }
061
062                 // If the provider is not already being launched, then get it
063                 // started.
064                 if (i >= N) {
065                     final long origId = Binder.clearCallingIdentity();
066                     ProcessRecord proc = startProcessLocked(cpi.processName,
```

```
067                         cpr.appInfo, false, 0, "content provider",
068                         new ComponentName(cpi.applicationInfo.packageName,
069                         cpi.name), false);
070                     ......
071                     mLaunchingProviders.add(cpr);
072                     ......
073                 }
```

ActivityManagerService把所有正在启动的Content Provider组件保存在它的一个成员变量mLaunchingProviders中。在启动ContentProviderRecord对象cpr所描述的Content Provider组件之前，第54行到第60行代码首先在成员变量mLaunchingProviders中检查它是否正处于启动的过程中。如果是，那么ActivityManagerService需要做的事情就是等待它启动完成；否则，第66行到第69行代码就会调用成员函数startProcessLocked来创建一个新的应用程序进程。等到新创建的应用程序进程启动完成之后，ActivityManagerService就可以请求它启动使用ContentProviderRecord对象cpr来描述的一个Content Provider组件了。第71行代码将ContentProviderRecord对象cpr保存在成员变量mLaunchingProviders中，表示它所描述的Content Provider组件正在启动中。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
074                 // Make sure the provider is published (the same provider class
075                 // may be published under multiple names).
076                 if (firstClass) {
077                     mProvidersByClass.put(cpi.name, cpr);
078                 }
079                 cpr.launchingApp = proc;
080                 mProvidersByName.put(name, cpr);
081
082                 ......
083             }
084         }
```

新创建的应用程序进程使用ProcessRecord对象proc来描述，第79行代码将它保存在ContentProviderRecord对象cpr的成员变量launchingApp中，表示它所描述的Content Provider组件是在ProcessRecord对象proc所描述的应用程序进程中启动的。此外，第77行和第80行代码还分别将ContentProviderRecord对象cpr保存在ActivityManagerService类的成员变量mProvidersByClass和mProvidersByName中。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
085         // Wait for the provider to be published...
086         synchronized (cpr) {
087             while (cpr.provider == null) {
088                 ......
089                 try {
090                     cpr.wait();
091                 } catch (InterruptedException ex) {
092                 }
093             }
094         }
095
096         return cpr;
097     }
098
099     ......
100 }
```

由于ContentProviderRecord对象cpr所描述的Content Provider组件是需要在新创建的应用程序进程中启动的，因此，ActivityManagerService就需要等待这个新创建的应用程序进程启动完成之后，才可

以请求它启动这个Content Provider组件。新创建的应用程序进程启动完成ContentProviderRecord对象cpr所描述的Content Provider组件之后，就会将它的一个代理对象传递给ActivityManagerService，这时候ActivityManagerService就会将这个代理对象保存在与它对应的一个ContentProviderRecord对象，即这里的ContentProviderRecord对象cpr的成员变量provider中。

第87行到第93行的while循环的作用就是等待ContentProviderRecord对象cpr所描述的Content Provider组件启动完成，并且将它的一个代理对象传递给ActivityManagerService。ActivityManagerService一旦获得了这个代理对象，就可以将它返回给需要访问它的应用程序进程了。

接下来，我们继续分析ContentProviderRecord对象cpr所描述的Content Provider组件的启动过程。

Step 9：ActivityManagerService.startProcessLocked

第9步可以参考7.2小节中的第23步，它主要就是调用Process类的静态成员函数start来创建一个新的应用程序进程，并且将这个新创建的应用程序进程的入口设置为ActivityThread类的静态成员函数main。

以上3步都是在ActivityManagerService中执行的，接下来第10步和第11步是在新创建的应用程序进程中执行的，主要是用来执行初始化进程的工作，如图10-9所示。

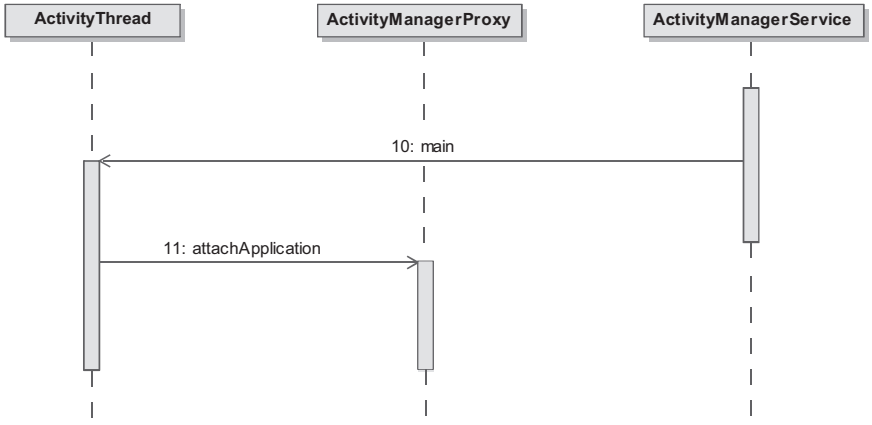


图10-9　ArticlesProvider组件的启动过程——Step 10和Step 11

Step 10：ActivityThread.main

Step 11：ActivityManagerProxy.attachApplication

第10步和第11步可以参考7.2小节中的第24步和第25步。其中，第10步会在新创建的应用程序进程中创建一个ActivityThread对象，以及一个ApplicationThread对象；而第11步向ActivityManagerService发出一个类型为ATTACH_APPLICATION_TRANSACTION的进程间通信请求，以及将前面创建的一个ApplicationThread对象传递给ActivityManagerService，以便ActivityManagerService可以和这个新创建的应用程序进程执行Binder进程间通信。

以上两步都是在新创建的应用程序进程中执行的，接下来第12步到第14步是在ActivityManagerService中执行的，主要是用来处理新的应用程序进程发出的类型为ATTACH_APPLICATION_TRANSACTION的进程间通信请求，如图10-10所示。

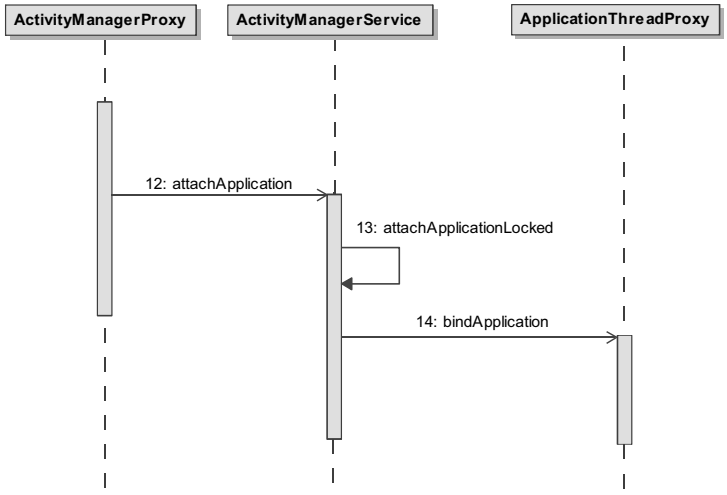


图10–10　ArticlesProvider组件的启动过程——Step 12至Step 14

Step 12：ActivityManagerService.attachApplication

第12步可以参考7.2小节中的第26步，它调用了ActivityManagerService类的成员函数attachApplicationLocked来处理类型为ATTACH_APPLICATION_TRANSACTION的进程间通信请求，用来执行启动ArticlesProvider组件的操作。

Step 13：ActivityManagerService.attachApplicationLocked

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     private final boolean attachApplicationLocked(IApplicationThread thread,
06             int pid) {
07         // Find the application record that is being attached...  either via
08         // the pid if we are running in multiple processes, or just pull the
09         // next app record if we are emulating process with anonymous threads.
10         ProcessRecord app;
11         if (pid != MY_PID && pid >= 0) {
12             synchronized (mPidsSelfLocked) {
13                 app = mPidsSelfLocked.get(pid);
14             }
15         }
16         ......
17
18         app.thread = thread;
19         ......
20
21         boolean normalMode = mProcessesReady || isAllowedWhileBooting(app.info);
22         List providers = normalMode ? generateApplicationProvidersLocked(app) : null;
23
24         try {
25             ......
26
27             thread.bindApplication(processName, app.instrumentationInfo != null
28                 ? app.instrumentationInfo : app.info, providers,
29                 app.instrumentationClass, app.instrumentationProfileFile,
30                 app.instrumentationArguments, app.instrumentationWatcher, testMode,
31                 isRestrictedBackupMode || !normalMode,
32                 mConfiguration, getCommonServicesLocked());
```

```
33
34              ......
35          } catch (Exception e) {
36              ......
37          }
38          ......
39
40          return true;
41      }
42
43      ......
44  }
```

参数pid指向的是前面所创建的应用程序进程的PID。在前面的Step 8中，ActivityManagerService以这个PID为关键字将一个ProcessRecord对象保存在了成员变量mPidsSelfLocked中，因此，第10行到第15行代码就可以通过参数pid将这个ProcessRecord对象取回来，并且保存在变量app中。

前面得到的ProcessRecord对象app就是用来描述新创建的应用程序进程的，现在既然这个应用程序进程已经启动起来了，第18行代码就将它的成员变量thread的值设置为参数thread所指向的一个ApplicationThread代理对象，这样，ActivityManagerService以后就可以通过这个ApplicationThread代理对象来和新创建的应用程序进程进行通信了。

第22行调用成员函数generateApplicationProvidersLocked来获得需要在ProcessRecord对象app所描述的应用程序进程中启动的Content Provider组件，它的实现如下所示。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02          implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03      ......
04
05      private final List generateApplicationProvidersLocked(ProcessRecord app) {
06          List providers = null;
07          try {
08              providers = AppGlobals.getPackageManager().
09                  queryContentProviders(app.processName, app.info.uid,
10                  STOCK_PM_FLAGS | PackageManager.GET_URI_PERMISSION_PATTERNS);
11          } catch (RemoteException ex) {
12          }
13          if (providers != null) {
14              final int N = providers.size();
15              for (int i=0; i<N; i++) {
16                  ProviderInfo cpi =
17                      (ProviderInfo)providers.get(i);
18                  ContentProviderRecord cpr = mProvidersByClass.get(cpi.name);
19                  if (cpr == null) {
20                      cpr = new ContentProviderRecord(cpi, app.info);
21                      mProvidersByClass.put(cpi.name, cpr);
22                  }
23                  app.pubProviders.put(cpi.name, cpr);
24                  app.addPackage(cpi.applicationInfo.packageName);
25                  ensurePackageDexOpt(cpi.applicationInfo.packageName);
26              }
27          }
28          return providers;
29      }
30
31      ......
32  }
```

第8行到第10行代码首先到PackageManagerService中去获得需要在ProcessRecord对象app所描述的应用程序进程中启动的Content Provider组件，并且保存在列表providers中。接着第15行到第26行的for

循环检查ActivityManagerService是否已经为这些Content Provider组件创建过一个ContentProviderRecord对象。如果没有，那么第20行代码就会分别为它们创建一个ContentProviderRecord对象，并且以它们的类名为关键字保存在ActivityManagerService类的成员变量mProvidersByClass所描述的一个HashMap中。从这里就可以看出，一个应用程序进程在启动时，会将需要在它里面运行的所有Content Provider组件都启动起来。

回到ActivityManagerService类的成员函数attachApplicationLocked中，获得了需要在ProcessRecord对象app所描述的应用程序进程中启动的Content Provider组件列表之后，接下来第27行到第32行代码就请求新创建的应用程序进程将这个列表中的Content Provider组件启动起来。

注意

这个Content Provider组件列表中包含了在前面Step 8中需要启动的ArticlesProvider组件。

Step 14：ApplicationThreadProxy.bindApplication

frameworks/base/core/java/android/app/ApplicationThreadNative.java

```
01 class ApplicationThreadProxy implements IApplicationThread {
02     ......
03
04     public final void bindApplication(String packageName, ApplicationInfo info,
05             List<ProviderInfo> providers, ComponentName testName,
06             String profileName, Bundle testArgs, IInstrumentationWatcher testWatcher, int debugMode,
07             boolean restrictedBackupMode, Configuration config,
08             Map<String, IBinder> services) throws RemoteException {
09         Parcel data = Parcel.obtain();
10         data.writeInterfaceToken(IApplicationThread.descriptor);
11         data.writeString(packageName);
12         info.writeToParcel(data, 0);
13         data.writeTypedList(providers);
14         if (testName == null) {
15             data.writeInt(0);
16         } else {
17             data.writeInt(1);
18             testName.writeToParcel(data, 0);
19         }
20         data.writeString(profileName);
21         data.writeBundle(testArgs);
22         data.writeStrongInterface(testWatcher);
23         data.writeInt(debugMode);
24         data.writeInt(restrictedBackupMode ? 1 : 0);
25         config.writeToParcel(data, 0);
26         data.writeMap(services);
27         mRemote.transact(BIND_APPLICATION_TRANSACTION, data, null,
28                 IBinder.FLAG_ONEWAY);
29         data.recycle();
30     }
31
32     ......
33 }
```

第10行到第26行代码将前面传进来的参数写入到Parcel对象data中，接着再通过ApplicationThreadProxy类内部的一个Binder代理对象mRemote向新创建的应用程序进程发送一个类型为BIND_APPLICATION_TRANSACTION的进程间通信请求。

以上3步都是在ActivityManagerService中执行的，接下来第15步到第24步是在新创建的应用程序进程中执行的，主要是用来处理ActivityManagerService发出的类型为BIND_APPLICATION_TRANSACTION

的进程间通信请求，如图10–11所示。

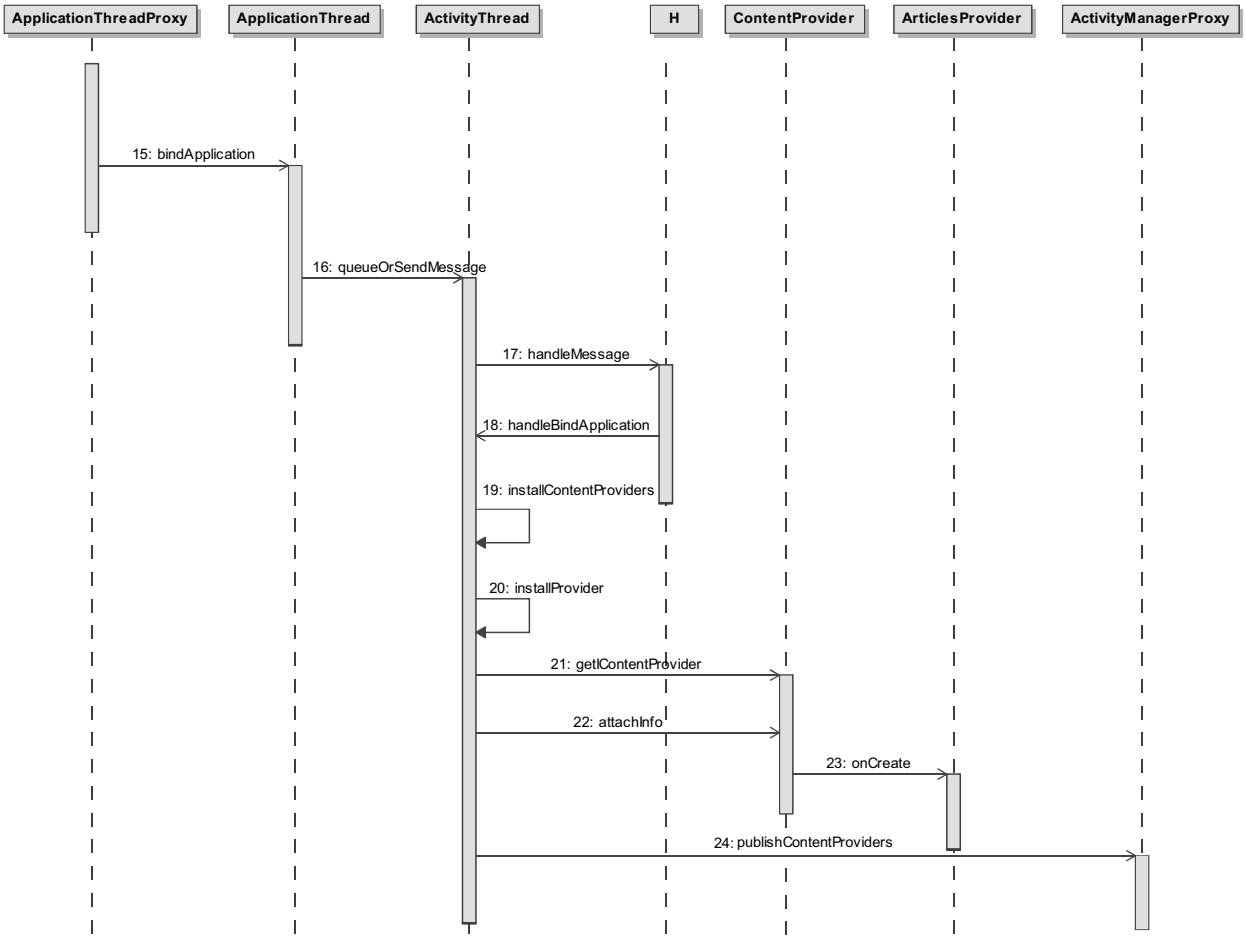


图10–11　ArticlesProvider组件的启动过程——Step 15至Step 24

Step 15：ApplicationThread.bindApplication

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final class ApplicationThread extends ApplicationThreadNative {
05         ......
06
07         public final void bindApplication(String processName,
08                 ApplicationInfo appInfo, List<ProviderInfo> providers,
09                 ComponentName instrumentationName, String profileFile,
10                 Bundle instrumentationArgs, IInstrumentationWatcher instrumentationWatcher,
11                 int debugMode, boolean isRestrictedBackupMode, Configuration config,
12                 Map<String, IBinder> services) {
13             ......
14
15             AppBindData data = new AppBindData();
16             data.processName = processName;
17             data.appInfo = appInfo;
18             data.providers = providers;
19             data.instrumentationName = instrumentationName;
20             data.profileFile = profileFile;
21             data.instrumentationArgs = instrumentationArgs;
22             data.instrumentationWatcher = instrumentationWatcher;
```

```
23              data.debugMode = debugMode;
24              data.restrictedBackupMode = isRestrictedBackupMode;
25              data.config = config;
26              queueOrSendMessage(H.BIND_APPLICATION, data);
27          }
28
29          ......
30      }
31
32      ......
33  }
```

ApplicationThread类的成员函数bindApplication用来处理类型为BIND_APPLICATION_TRANSACTION的进程间通信请求。

第15行到第26行代码首先将要启动的Content Proivder组件列表封装成一个AppBindData对象，然后再调用ActivityThread类的成员函数queueOrSendMessage将这个AppBindData对象封装成一个类型为BIND_APPLICATION的消息，以便可以发送到新创建的应用程序进程的主线程的消息队列中。

Step 16：ActivityThread.queueOrSendMessage

第16步可以参考7.2小节中的第31步，它主要是将上一步创建的AppBindData对象封装成一个类型为BIND_APPLICATION的消息，然后调用ActivityThread类的成员变量mH的成员函数sendMessage将这个消息发送到新创建的应用程序进程的主线程的消息队列中。这个消息最终是在ActivityThread类的成员变量mH的成员函数handleMessage中处理的。

Step 17：H.handleMessage

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final class H extends Handler {
05         ......
06
07         public void handleMessage(Message msg) {
08             ......
09
10             switch (msg.what) {
11                 ......
12                 case BIND_APPLICATION:
13                     AppBindData data = (AppBindData)msg.obj;
14                     handleBindApplication(data);
15                     break;
16                 ......
17             }
18
19             ......
20         }
21
22         ......
23     }
24
25     ......
26 }
```

第13行代码首先将Message对象msg的成员变量obj转换成一个AppBindData对象，它里面包含了要启动的Content Provider组件列表。接着第14行调用ActivityThread类的成员函数handleBindApplication将这些Content Provider组件启动起来。

Step 18：ActivityThread. handleBindApplication

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final void handleBindApplication(AppBindData data) {
05         ......
06
07         List<ProviderInfo> providers = data.providers;
08         if (providers != null) {
09             installContentProviders(app, providers);
10             ......
11         }
12
13         ......
14     }
15
16     ......
17 }
```

AppBindData 对象data的成员变量providers保存了要在当前应用程序进程中启动的Content Provider组件，接下来第9行就会调用ActivityThread类的成员函数installContentProviders来启动这些Content Provider组件。

Step 19：ActivityThread.installContentProviders

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final void installContentProviders(
05             Context context, List<ProviderInfo> providers) {
06         final ArrayList<IActivityManager.ContentProviderHolder> results =
07             new ArrayList<IActivityManager.ContentProviderHolder>();
08
09         Iterator<ProviderInfo> i = providers.iterator();
10         while (i.hasNext()) {
11             ProviderInfo cpi = i.next();
12             ......
13             IContentProvider cp = installProvider(context, null, cpi, false);
14             if (cp != null) {
15                 IActivityManager.ContentProviderHolder cph =
16                     new IActivityManager.ContentProviderHolder(cpi);
17                 cph.provider = cp;
18                 results.add(cph);
19                 ......
20             }
21         }
22
23         try {
24             ActivityManagerNative.getDefault().publishContentProviders(
25                 getApplicationThread(), results);
26         } catch (RemoteException ex) {
27         }
28     }
29
30     ......
31 }
```

第10行到第21行的while循环首先调用成员函数installProvider在当前应用程序进程中启动保存在

列表providers中的每一个Content Provider组件。这些Content Provider组件启动起来之后，我们就可以获得它们的一个IContentProvider接口。接着第15行到第17行代码将这个IContentProvider接口封装成一个ContentProviderHolder对象。最后第24行和第25行代码调用ActivityManagerService代理对象的成员函数publishContentProviders将这些ContentProviderHolder对象传递给ActivityManagerService，以便ActivityManagerService可以获得它们所描述的Content Provider组件的一个访问接口。

接下来，我们首先分析ActivityThread类的成员函数installProvider在当前应用程序进程中启动一个Content Provider组件的过程，然后分析ActivityManagerService代理对象的成员函数publishContentProviders将启动完成之后的Content Provider组件发布到ActivityManagerService中的过程。

Step 20：ActivityThread.installProvider

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final IContentProvider installProvider(Context context,
05             IContentProvider provider, ProviderInfo info, boolean noisy) {
06         ContentProvider localProvider = null;
07         if (provider == null) {
08             ......
09
10             Context c = null;
11             ApplicationInfo ai = info.applicationInfo;
12             if (context.getPackageName().equals(ai.packageName)) {
13                 c = context;
14             }
15             ......
16
17             try {
18                 final java.lang.ClassLoader cl = c.getClassLoader();
19                 localProvider = (ContentProvider)cl.
20                     loadClass(info.name).newInstance();
21                 provider = localProvider.getIContentProvider();
22                 ......
23
24                 // XXX Need to create the correct context for this provider.
25                 localProvider.attachInfo(c, info);
26             } catch (java.lang.Exception e) {
27                 ......
28             }
29
30         }
31         ......
32
33         synchronized (mProviderMap) {
34             // Cache the pointer for the remote provider.
35             String names[] = PATTERN_SEMICOLON.split(info.authority);
36             for (int i=0; i<names.length; i++) {
37                 ProviderClientRecord pr = new ProviderClientRecord(names[i], provider,
38                     localProvider);
39                 try {
40                     ......
41                     mProviderMap.put(names[i], pr);
42                 } catch (RemoteException e) {
43                     return null;
44                 }
45             }
46             if (localProvider != null) {
47                 mLocalProviders.put(provider.asBinder(),
48                     new ProviderClientRecord(null, provider, localProvider));
```

```
49              }
50          }
51
52          return provider;
53      }
54
55      ......
56  }
```

前面传进来的参数provider为null，表示要在当前应用程序进程中将参数info所描述的Content Provider组件启动起来；另外一个参数context描述了要启动的Content Provider组件的运行上下文环境。

参数info的成员变量name描述了要在当前应用程序进程中启动的Content Provider组件的类名。第18行到第20行代码根据这个类名在当前应用程序进程中创建一个Content Provider组件实例，并且将它保存在ContentProvider对象localProvider中。

第21行调用ContentProvider对象localProvider的成员函数getIContentProvider来获得它的一个IContentProvider接口provider。这个IContentProvider接口最终是需要发布到ActivityManagerService中的，以便ActivityManagerService可以将它返回给那些需要访问它所描述的Content Provider组件的应用程序进程。接下来第25行调用ContentProvider对象localProvider的成员函数attachInfo来进一步初始化前面所创建的Content Provider组件。

参数info的成员变量authority描述了前面所创建的Content Provider组件的android:authorities属性值。Content Provider组件的android:authorities属性是一个多值属性，第36行到第45行的for循环为它的每一个android:authorities属性值创建一个ProviderClientRecord对象，并且将它们保存在ActivityThread类的成员变量mProviderMap所描述的一个HashMap中。

对一个应用程序进程来说，它所访问的Content Provider组件有可能是在当前应用程序进程中运行的，也有可能是在其他应用程序进程中运行的。为了区分这两种情况，ActivityThread类使用成员变量mLocalProviders所描述的一个HashMap来保存那些在当前应用程序进程中运行的Content Provider组件。从前面的执行过程可以知道，如果一个Content Provider组件是在当前应用程序进程中运行的，那么变量localProvider的值就不等于null。因此，第47行和第48行代码就首先将这个Content Provider组件封装成一个ProviderClientRecord对象，然后再将它保存在ActivityThread类的成员变量mLocalProviders中。

至此，一个Content Provider组件就在一个应用程序进程中启动起来了，最后第52行将它的一个IContentProvider接口返回给调用者。

接下来，我们继续分析Content Provider组件的成员函数getIContentProvider和attachInfo的实现，以便可以了解一个Content Provider组件的IContentProvider接口的获取过程，以及初始化过程。

Step 21：ContentProvider.getIContentProvider

frameworks/base/core/java/android/content/ContentProvider.java

```
01 public abstract class ContentProvider implements ComponentCallbacks {
02     ......
03
04     private Transport mTransport = new Transport();
05     ......
06
07     class Transport extends ContentProviderNative {
08         ......
09     }
10
11     public IContentProvider getIContentProvider() {
12         return mTransport;
13     }
14
15     ......
16 }
```

每一个Content Provider组件内部都有一个类型为Transport的Binder本地对象，这个Binder本地对象是用来传递给ActivityManagerService的，然后ActivityManagerService再将引用了它的一个Binder代理对象返回给需要访问该Content Provider组件的其他应用程序进程。这样，其他应用程序进程就可以通过这个Binder代理对象来间接地访问一个Content Provider组件中的数据了。

ContentProvider类的成员函数getIContentProvider的实现很简单，它只是将其成员变量mTransport所描述的一个Transport对象的一个IContentProvider接口返回给调用者。

Step 22：ContentProvider.attachInfo

frameworks/base/core/java/android/content/ContentProvider.java

```
01 public abstract class ContentProvider implements ComponentCallbacks {
02     ......
03
04     public void attachInfo(Context context, ProviderInfo info) {
05         /*
06         * Only allow it to be set once, so after the content service gives
07         * this to us clients can't change it.
08         */
09         if (mContext == null) {
10             mContext = context;
11             ......
12             if (info != null) {
13                 setReadPermission(info.readPermission);
14                 setWritePermission(info.writePermission);
15                 setPathPermissions(info.pathPermissions);
16                 ......
17             }
18             ContentProvider.this.onCreate();
19         }
20     }
21
22     ......
23 }
```

ContentProvider类的成员函数attachInfo用来初始化前面所创建的一个Content Provider组件。

参数context用来描述要初始化的Content Provider组件的运行上下文环境，第10行代码将它保存在ContentProvider类的成员变量mContext中。

参数info的成员变量readPermission、writePermission和pathPermissions描述了要初始化的Content Provider组件的访问权限，第13行到第15行代码分别调用成员函数setReadPermission、setWritePermission和setPathPermissions将它们保存在ContentProvider类的内部。

最后，第18行调用了由ContentProvider的子类来重写的成员函数onCreate，以便这些自定义的Content Provider组件类可以在启动时有机会执行一些业务相关的初始化工作。

在我们这个场景中，正在初始化的Content Provider组件即为前面要启动的ArticlesProvider组件，因此，接下来就会调用它的成员函数onCreate来执行业务相关的初始化工作。

Step 23：ArticlesProvider.onCreate

packages/experimental/ArticlesProvider/src/shy/luo/providers/articles/ArticlesProvider.java

```
01 public class ArticlesProvider extends ContentProvider {
02     ......
03
04     @Override
05     public boolean onCreate() {
06         Context context = getContext();
07         resolver = context.getContentResolver();
08         dbHelper = new DBHelper(context, DB_NAME, null, DB_VERSION);
```

```
09
10          Log.i(LOG_TAG, "Articles Provider Create");
11          return true;
12      }
13
14      ......
15  }
```

由于一个Content Provider组件的成员函数onCreate是在启动的过程中被调用的，因此，我们要尽量避免在里面执行耗时的操作，例如与IO相关的操作；否则，就有可能造成这个Content Provider组件启动超时。

这一步执行完成之后，首先返回到前面的Step 20中，然后再返回到前面的Step 19中，即ActivityThread类的成员函数installContentProviders中，接下来就会调用ActivityManagerService代理对象的成员函数publishContentProviders将前面所启动的Content Provider组件的一个IContentProvider访问接口发布到ActivityManagerService中。

Step 24：ActivityManagerProxy. publishContentProviders

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
01 class ActivityManagerProxy implements IActivityManager
02 {
03     ......
04
05     public void publishContentProviders(IApplicationThread caller,
06                 List<ContentProviderHolder> providers) throws RemoteException
07     {
08         Parcel data = Parcel.obtain();
09         Parcel reply = Parcel.obtain();
10         data.writeInterfaceToken(IActivityManager.descriptor);
11         data.writeStrongBinder(caller != null ? caller.asBinder() : null);
12         data.writeTypedList(providers);
13         mRemote.transact(PUBLISH_CONTENT_PROVIDERS_TRANSACTION, data, reply, 0);
14         reply.readException();
15         data.recycle();
16         reply.recycle();
17     }
18
19     ......
20 }
```

第10行到第12行代码将前面传进来的参数写入到Parcel对象data中，接着再通过ActivityManagerProxy类内部的一个Binder代理对象mRemote向ActivityManagerService发送一个类型为PUBLISH_CONTENT_PROVIDERS_TRANSACTION的进程间通信请求。

以上10步都是在新创建的应用程序进程中执行的，接下来第25步是在ActivityManagerService中执行的，主要是用来处理新创建的应用程序进程发出的类型为PUBLISH_CONTENT_PROVIDERS_TRANSACTION的进程间通信请求，如图10-12所示。

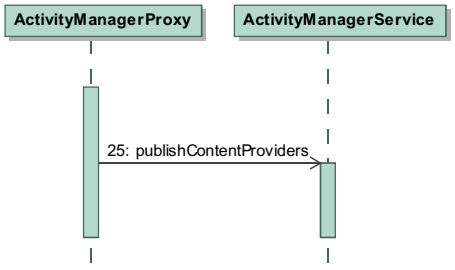


图10-12　ArticlesProvider组件的启动过程——Step 25

Step 25：ActivityMangerService.publishContentProviders

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     public final void publishContentProviders(IApplicationThread caller,
06             List<ContentProviderHolder> providers) {
07         ......
08
09         synchronized(this) {
10             final ProcessRecord r = getRecordForAppLocked(caller);
11             ......
12
13             final int N = providers.size();
14             for (int i=0; i<N; i++) {
15                 ContentProviderHolder src = providers.get(i);
16                 ......
17                 ContentProviderRecord dst = r.pubProviders.get(src.info.name);
18                 if (dst != null) {
19                     ......
20
21                     int NL = mLaunchingProviders.size();
22                     int j;
23                     for (j=0; j<NL; j++) {
24                         if (mLaunchingProviders.get(j) == dst) {
25                             mLaunchingProviders.remove(j);
26                             j--;
27                             NL--;
28                         }
29                     }
30                     synchronized (dst) {
31                         dst.provider = src.provider;
32                         dst.app = r;
33                         dst.notifyAll();
34                     }
35                     ......
36                 }
37             }
38         }
39     }
40
41     ......
42 }
```

ActivityManagerService类的成员函数publishContentProviders用来处理类型为PUBLISH_CONTENT_PROVIDERS_TRANSACTION的进程间通信请求。

参数caller是一个类型为ApplicationThread的Binder代理对象，它引用了运行在新创建的应用程序进程中的一个ApplicationThread对象，第10行代码通过它来获得用来描述新创建的应用程序进程的一个ProcessRecord对象r。

新创建的应用程序进程在启动时，会将需要在它里面运行的Content Provider组件启动起来。从前面的Step 13可以知道，这些Content Provider组件在ActivityManagerService中使用一个ContentProviderRecord对象来描述，它们保存在用来描述新创建的应用程序进程的一个ProcessRecord对象r的成员变量pubProviders中。

参数providers包含了要发布到ActivityManagerService中的Content Provider组件，每一个Content Provider组件都使用一个ContentProviderHolder对象来描述，它里面包含了要发布的Content Provider组件

的一个IContentProvider接口。

第14行到第37行的for循环首先取出保存在参数providers中的每一个ContentProviderHolder对象src，然后在ActivityManagerService中找到与它对应的一个ContentProviderRecord对象dst，最后将ContentProviderHolder对象src所描述的一个Content Provider组件的一个IContentProvider访问接口保存在ContentProviderRecord对象dst的成员变量provider中，如第31行代码所示。

从前面的Step 8可以知道，一个应用程序进程在请求ActivityManagerService返回一个Content Provider组件的代理对象时，如果这个Content Provider组件还没有启动起来，那么ActivityManagerService就会先创建一个新的应用程序进程来启动该Content Provider组件，然后再在一个while循环中等待该Content Provider组件启动完成，并且将它的一个代理对象发布到ActivityManagerService中。现在既然这个Content Provider已经启动完成，并且将它的一个代理对象，即一个类型为Transport的Binder代理对象发布到ActivityManagerService中了，因此，前面正在等待的一个ActivityManagerService线程就可以停止等待，并且将这个类型为Transport的Binder代理对象封装成一个ContentProviderHolder对象返回给请求它的应用程序进程。

这一步执行完成之后，就会使得ActivityManagerService从前面的Step 8返回Step 5中，即返回到MainActivity组件所运行在的应用程序进程中，继续执行ActivityThread类的成员函数installProvider，用来保存前面从ActivityManagerService获得的一个Content Provider组件的一个IContentProvider访问接口，如图10-13所示。

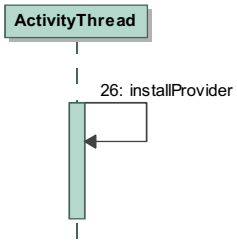


图10-13　ArticlesProvider组件的启动过程——Step 26

Step 26：ActivityThread.installProvider

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final IContentProvider installProvider(Context context,
05             IContentProvider provider, ProviderInfo info, boolean noisy) {
06         ......
07         if (provider == null) {
08             ......
09         }
10         ......
11
12         synchronized (mProviderMap) {
13             // Cache the pointer for the remote provider.
14             String names[] = PATTERN_SEMICOLON.split(info.authority);
15             for (int i=0; i<names.length; i++) {
16                 ProviderClientRecord pr = new ProviderClientRecord(names[i], provider,
17                     localProvider);
18                 try {
19                     provider.asBinder().linkToDeath(pr, 0);
20                     mProviderMap.put(names[i], pr);
21                 } catch (RemoteException e) {
22                     return null;
```

```
                }
            }
            ......
        }

        return provider;
    }

    ......
}
```

这一步与前面的Step 20一样，都是执行ActivityThread类的成员函数installProvider。不过，前面的Step 20是在启动ArticlesProvider组件的应用程序进程中执行的，而这一步是在访问ArticlesProvider组件的MainActivity组件所运行在的应用程序进程中执行的。另外一个区别是，这一步得到的参数provider不等于null，它用来描述一个在其他应用程序进程中启动的Content Provider组件的一个IContentProvider访问接口。

这一步执行完成之后，就返回到前面的Step 1中，这时候MainActivity组件就获得了与URI 值“content://shy.luo.providers.articles/item”相对应的ArticlesProvider组件的一个IContentProvider访问接口，最后就可以通过这个接口来访问ArticlesProvider组件中的博客文章信息。

接下来，我们继续分析MainActivity组件访问运行在另外一个应用程序进程中的ArticlesProvider组件的博客文章信息的过程，即分析Content Provider组件的数据共享原理。

10.3　Content Provider组件的数据共享原理

前面提到，在Android系统中，由于权限控制，不同的应用程序是不能直接访问对方的数据的。如果需要不同的应用程序之间进行数据共享，那么就可以利用Content Provider组件来实现。

Content Provider组件通过Binder进程间通信机制来突破以应用程序为边界的权限控制，同时，它又以匿名共享内存作为数据传输媒介，从而提供了一种高效的数据共享方式。

注意

虽然Binder进程间通信机制可以突破以应用程序为边界的权限控制，但是并不意味着一个应用程序可以随意访问另外一个应用程序的数据，因为数据的所有者可以在内部实施数据访问策略，即可以控制哪些数据是可访问的，以及控制哪些数据可以由谁来访问。

Content Provider组件将要传输的共享数据抽象为一个游标，因此，我们在实现自己的Content Provider组件时，需要提供一个游标，以便可以将要共享的数据从一个应用程序传递给另外一个应用程序访问。

接下来，我们就以前面10.1小节中的应用程序为例，来分析Content Provider组件的数据共享原理。其中，10.3.1小节介绍Content Provider组件的数据共享模型，10.3.2小节分析Content Provider组件传输共享数据的过程。

10.3.1　数据共享模型

在前面的10.1小节中，当MainActivity组件请求ArticlesProvider组件返回其内部的博客文章列表时，ArticlesProvider组件首先将其内部的博客文章列表封装在一个SQLite数据库游标中，然后再返回给MainActivity组件。

由于ArticlesProvider组件内部的博客文章列表本身就是保存在一个SQLite数据库中的，因此，当这些博客文章被读取出来时，它们就很自然地保存在一个SQLite数据库游标中。SQLite数据库游标是由SQLiteCursor类来实现的，本节就通过它来介绍Content Provider组件的数据共享模型。

SQLiteCursor类的实现关系图如图10-14所示。

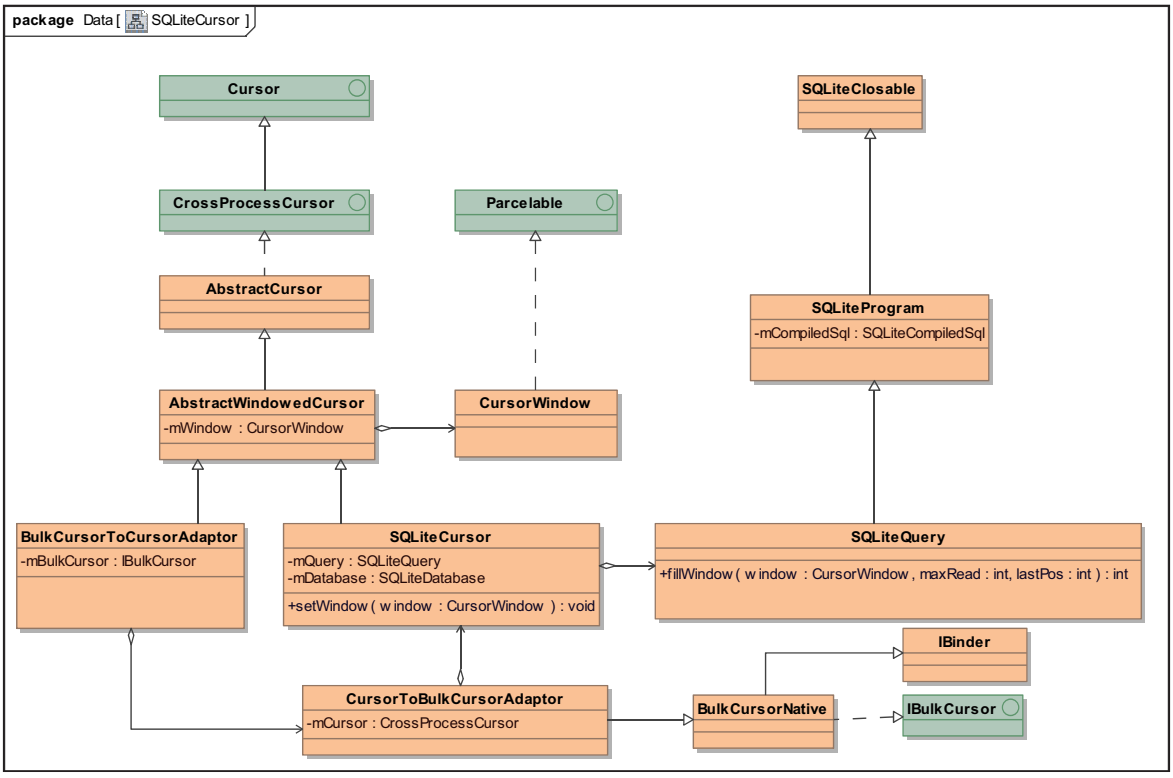


图10-14　SQLiteCurosr类的实现关系图

MainActivity组件在请求ArticlesProvider组件返回其内部的博客文章信息之前，首先会在当前应用程序进程中创建一个CursorWindow对象。CursorWindow类实现了Parcelable接口，并且在内部包含了一块匿名共享内存。接下来MainActivity组件就会通过Binder进程间通信机制将前面所创建的CursorWindow对象（连同它内部的匿名共享内存）传递给ArticlesProvider组件。

ArticlesProvider组件获得了MainActivity组件发送过来的CursorWindow对象之后，就会创建一个SQLiteCursor对象。SQLiteCursor类继承了AbstractWindowedCursor类，而AbstractWindowedCursor类又继承了AbstractCursor类。AbstractCursor类实现了CrossProcessCursor接口，而CrossProcessCursor接口又是从Cursor接口继承下来的。因此，SQLiteCursor类可以用来描述一个游标，即可以用来传输Content Provider组件中的共享数据。

AbstractWindowedCursor类有一个类型为CursorWindow的成员变量mWindow。ArticlesProvider组件在创建SQLiteCursor对象时，会调用它的成员函数setWindow将MainActivity组件发送过来的CursorWindow对象保存在其父类AbstractWindowedCursor的成员变量mWindow中。

SQLiteCursor对象创建完成之后，ArticlesProvider组件就会将MainActivity组件所请求的博客文章信息保存在这个SQLiteCursor对象中，实际上是保存在与它所关联的一个CursorWindow对象的内部的一块匿名共享内存中。由于MainActivity组件是可以访问这块匿名共享内存的，因此，它就可以通过这块匿名共享内存来获得ArticlesProvider组件返回给它的博客文章信息。

由于SQLiteCursor对象并不是一个Binder本地对象，因此，ArticlesProvider组件不能直接将它返

回给MainActivity组件使用。ArticlesProvider组件首先会创建一个CursorToBulkCursorAdaptor对象，用来适配前面所创建的一个SQLiteCursor对象，即将这个SQLiteCursor对象保存在它的成员变量mCursor中。CursorToBulkCursorAdaptor类继承了BulkCursorNative类，而BulkCursorNative类实现了IBulkCursor接口，并且继承了IBinder类，因此，它可以用来描述一个实现了IBulkCursor接口的Binder本地对象。ArticlesProvider组件接着就会将这个CursorToBulkCursorAdaptor对象返回给MainActivity组件。

MainActivity组件在请求ArticlesProvider组件返回其内部的博客文章信息之前，除了会创建一个CursorWindow对象之外，还会创建一个BulkCursorToCursorAdaptor对象，并且前面所创建的CursorWindow对象就是保存在它的父类AbstractWindowedCursor的成员变量mWindow中的。

MainActivity组件接收到ArticlesProvider组件返回来的CursorToBulkCursorAdaptor对象之后，实际上获得的是一个CursorToBulkCursorAdaptor代理对象，接着就会将它保存在之前所创建的一个BulkCursorToCursorAdaptor对象的成员变量mBulkCursor中。这时候MainActivity组件就可以通过这个BulkCursorToCursorAdaptor对象来读取ArticlesProvider组件返回来的博客文章信息了。

至此，我们就得到了一系列的对象，有的是在MainActivity组件这一侧创建的，有的是在ArticlesProvider组件这一侧创建的，它们互相联系在一起就得到了Content Provider组件的数据共享模型，如图10-15所示。

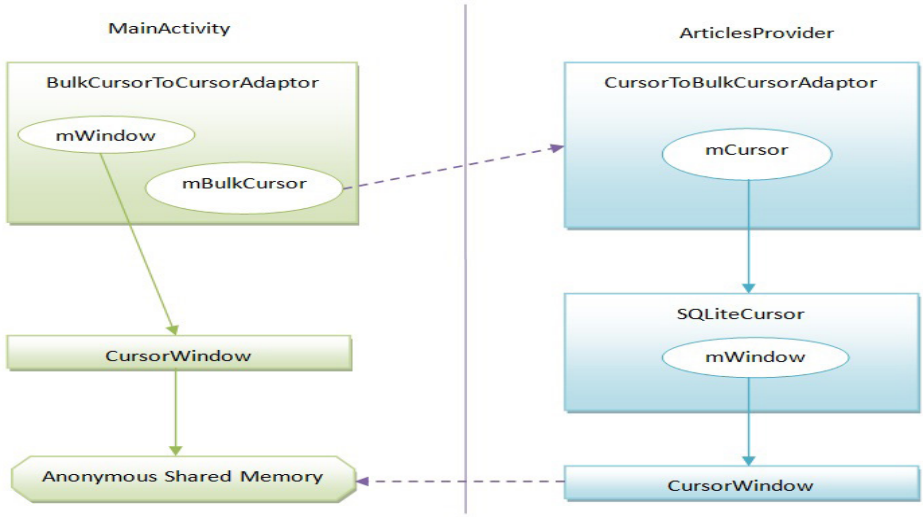


图10-15　Content Provider组件的数据共享模型

在图10-15所描述的Content Provider组件的数据共享模型中，我们可以看出：

（1）左边的BulkCursorToCursorAdaptor对象的成员变量mBulkCursor引用了右边的CursorToBulkCursorAdaptor对象。

（2）左边的CursorWindow对象和右边的CursorWindow对象引用了同一块匿名共享内存。

注意

这时候系统只是在MainActivity组件和ArticlesProvider组件之间建立了一个数据共享模型，ArticlesProvider组件还没有真正把MainActivity组件所请求的博客文章信息写入到它们所共享的一块匿名共享内存中，即右边的SQLiteCursor对象还没有将MainActivity组件所请求的博客文章信息从数据库中读取出来。

SQLiteCursor类有两个成员变量mDatabase和mQuery，它们的类型分别为SQLiteDatabase和SQLiteQuery，前者用来描述一个SQLite数据库，后者用来描述一个SQLite数据库查询计划。SQLiteQuery类继承了SQLiteProgram类，而SQLiteProgram类又继承了SQLiteClosable类。SQLiteProgram类内部有一个成员变量

mCompiledSql，它保存了一个已经编译好的SQL语句，相当于准备好了要从数据库mDatabase中获取的数据。

当MainActivity组件第一次通过左边的BulkCursorToCursorAdaptor对象来读取ArticlesProvider组件返回来的博客文章信息时，左边的BulkCursorToCursorAdaptor对象就会首先通过其成员变量mBulkCursor来请求右边的CursorToBulkCursorAdaptor对象将它所请求的博客文章信息写入到右边的CursorWindow对象所引用的一块匿名共享内存中。接着左边的BulkCursorToCursorAdaptor对象就可以通过其内部的CursorWindow对象来获得MainActivity组件所请求的博客文章信息了，因为左右两边的CursorWindow对象引用了同一块匿名共享内存。右边的CursorToBulkCursorAdaptor对象实际上是通过其内部的SQLiteCursor对象来获得MainActivity组件所请求的博客文章信息的，而这个SQLiteCursor对象又是通过调用其内部的数据库查询计划mQuery的成员函数fillWindow来从数据库mDatabase中获得MainActivity组件所请求的博客文章信息的。

注意

MainActivity组件以后再通过左边的BulkCursorToCursorAdaptor对象来读取ArticlesProvider组件返回来的博客文章信息时，左边的BulkCursorToCursorAdaptor对象就可以直接通过其内部的CursorWindow对象来获得MainActivity组件所请求的博客文章信息了。

如果MainActivity组件在请求ArticlesProvider组件返回其内部的博客文章信息时，要求ArticlesProvider组件返回符合条件的博客文章信息的元数据，例如，要求返回符合条件的博客文章信息的行数，即符合条件的博客文章的数量，那么ArticlesProvider组件在将右边的CursorToBulkCursorAdaptor对象返回给MainActivity组件之前，就会通过右边的SQLiteCursor对象从数据库mDatabase中获得MainActivity组件所请求的博客文章信息，并且保存在右边的CursorWindow对象所引用的一块匿名共享内存中。因为只有获得了这些博客文章信息之后，ArticlesProvider组件才能将它的元数据返回给MainActivity组件。

以上就是Content Provider组件的数据共享模型。接下来我们再以MainActivity组件获取ArticlesProvider组件中指定位置值的博客文章条目为例，详细分析Content Provider组件传输共享数据的过程，以便我们更好地理解Content Provider组件的数据共享模型。

10.3.2 数据传输过程

MainActivity组件在更新其用户界面上的博客文章列表时，需要调用其内部的一个ArticlesAdapter对象的成员函数getArticleByPos来逐条地获得ArticlesProvider组件中的博客文章条目，然后将它们显示在这个博客文章列表中。

MainActivity组件通过调用其内部的一个ArticlesAdapter对象的成员函数getArticleByPos，从ArticlesProvider组件中获得一个指定位置值的博客文章条目的过程如下所示。

（1）MainActivity组件通过Articles.CONTENT_POS_URI在当前应用程序进程中获得用来访问ArticlesProvider组件的一个Binder代理对象。

（2）MainActivity组件通过上一步获得的Binder代理对象来请求ArticlesProvider组件返回一个指定位置值的博客文章条目。

（3）ArticlesProvider组件创建一个SQLiteCursor对象，用来从内部的SQLite数据库中获得一个指定位置值的博客文章条目，然后将这个SQLiteCursor对象封装成一个CursorToBulkCursorAdaptor对象，以便可以传递给MainActivity组件。

（4）ArticlesProvider组件将上一步创建的CursorToBulkCursorAdaptor对象返回给MainActivity组件之前，发现MainActivity组件需要获得它所请求的博客文章信息的元数据，因此，ArticlesProvider

组件就会执行保存在上一步所创建的SQLiteCursor对象内部的一个数据库查询计划，以便可以获得MainActivity组件所请求的博客文章信息的元数据。

这个过程一共分为24个步骤，接下来我们就详细分析每一个步骤。

第1步到第6步的过程如图10-16所示。

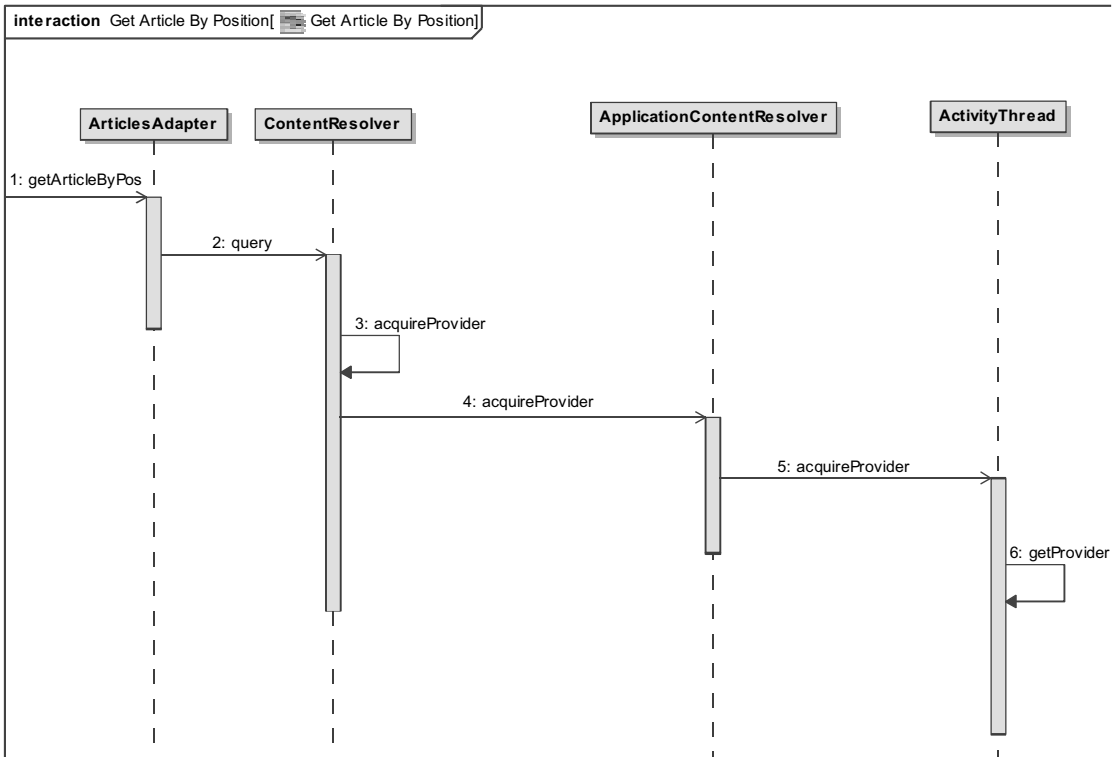


图10-16　ArticlesProvider组件传输数据的过程——Step 1至Step 6

Step 1：ArticlesAdapter.getArticleByPos

packages/experimental/Article/src/shy/luo/article/ArticlesAdapter.java

```
01 public class ArticlesAdapter {
02     ......
03
04     private ContentResolver resolver = null;
05
06     public ArticlesAdapter(Context context) {
07         resolver = context.getContentResolver();
08     }
09     ......
10
11     public Article getArticleByPos(int pos) {
12         Uri uri = ContentUris.withAppendedId(Articles.CONTENT_POS_URI, pos);
13
14         String[] projection = new String[] {
15             Articles.ID,
16             Articles.TITLE,
17             Articles.ABSTRACT,
18             Articles.URL
19         };
20
21         Cursor cursor = resolver.query(uri, projection, null, null, Articles.DEFAULT_SORT_ORDER);
22         if (!cursor.moveToFirst()) {
23             return null;
24         }
25
```

```
26          int id = cursor.getInt(0);
27          String title = cursor.getString(1);
28          String abs = cursor.getString(2);
29          String url = cursor.getString(3);
30
31          return new Article(id, title, abs, url);
32      }
33  }
```

参数pos是一个整数，第12行代码将它附加到常量Articles.CONTENT_POS_URI的后面，就得到了一个值为“content://shy.luo.providers.articles/pos/#”的URI。接着第21行调用ContentResolver对象resolver的成员函数query来请求与这个URI对应的Content Provider组件返回符合条件的数据，即请求ArticlesProvider组件返回第pos个位置的博客文章条目。ArticlesProvider组件返回来的博客文章条目可以通过游标cursor来访问，因此，第26行到第29行代码就从游标cursor中取出它的ID、标题、摘要和URL等信息，最后第31行将这些信息封装成一个Article对象返回给调用者。

下面我们就重点分析ContentResolver类的成员函数query的调用过程。

Step 2：ContentResolver.query

frameworks/base/core/java/android/content/ContentResolver.java

```
01 public abstract class ContentResolver {
02      ......
03
04      public final Cursor query(Uri uri, String[] projection,
05              String selection, String[] selectionArgs, String sortOrder) {
06          IContentProvider provider = acquireProvider(uri);
07          if (provider == null) {
08              return null;
09          }
10          try {
11              ......
12              Cursor qCursor = provider.query(uri, projection, selection, selectionArgs, sortOrder);
13              ......
14
15              return new CursorWrapperInner(qCursor, provider);
16          } catch (RemoteException e) {
17              ......
18          } catch(RuntimeException e) {
19              ......
20          }
21      }
22
23      ......
24  }
```

第6行首先调用成员函数acquireProvider来获得与参数uri对应的Content Provider组件的一个IContentProvider访问接口，接着第12行调用这个接口的成员函数query来请求与它对应的Content Provider组件返回符合条件的数据。

接下来，我们首先分析ContentResolver类的成员函数acquireProvider的调用过程，然后分析IContentProvider接口的成员函数query的实现。

Step 3：ContentResolver.acquireProvider

Step 4：ApplicationContentResolver.acquireProvider

Step 5：ActivityThread.acquireProvider

Step 6：ActivityThread.getProvider

第3步到第6步可以参考10.2小节中的第2步到第5步，它们主要是获得与 “content://shy.luo.providers.

articles/pos/#”对应的一个Content Provider组件代理对象，并且将这个代理对象的一个IContentProvider接口返回给调用者。

从前面10.2小节的内容可以知道，值为“content://shy.luo.providers.articles/pos/#”的URI对应的Content Provider组件即为ArticlesProvider组件。ArticlesProvider组件在一个独立的应用程序进程中启动完成之后，会将内部的一个Transport对象发布到ActivityManagerService中，以便其他应用程序可以通过这个Transport对象的代理对象来访问ArticlesProvider组件。

返回到前面的Step 2中，即ContentResolver类的成员函数query中，从前面的调用过程可以知道，IContentProvider接口provider指向了ArticlesProvider组件内部的一个Transport对象的代理对象，即一个ContentProviderProxy对象。

接下来第7步到第11步主要用来调用ContentProviderProxy类的成员函数query来创建和初始化一个CursorWindow对象，以便可以将它传递给ArticlesProvider组件，用来保存符合条件的博客文章信息，如图10–17所示。

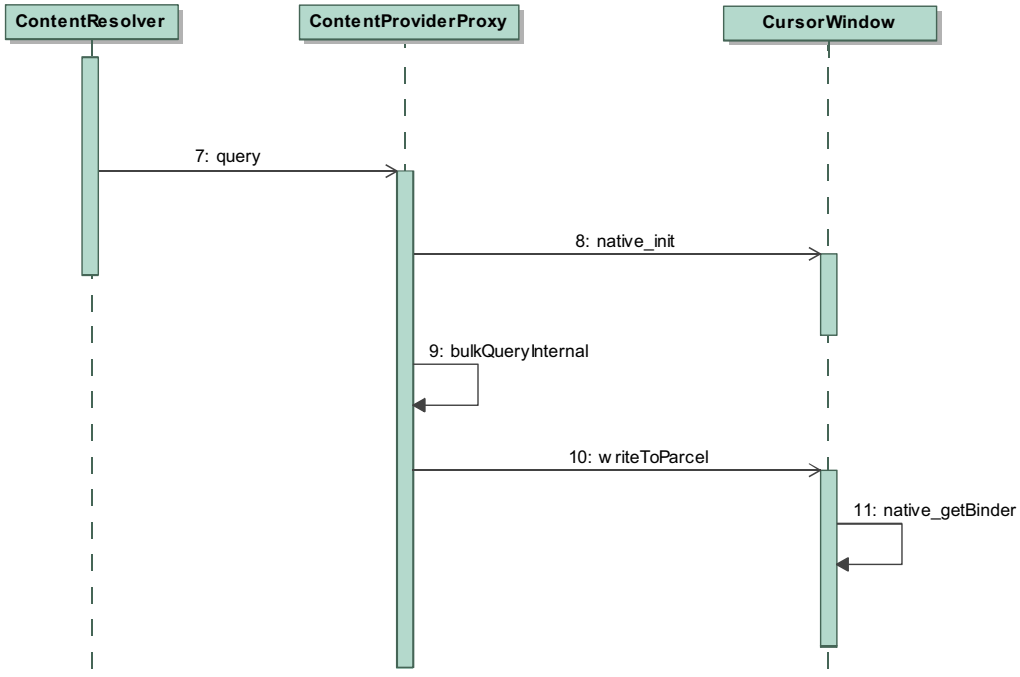


图10–17　ArticlesProvider组件传输数据的过程——Step 7至Step 11

Step 7：ContentProviderProxy.query

frameworks/base/core/java/android/content/ContentProviderNative.java

```
01 final class ContentProviderProxy implements IContentProvider {
02     ......
03
04     public Cursor query(Uri url, String[] projection, String selection,
05             String[] selectionArgs, String sortOrder) throws RemoteException {
06         //TODO make a pool of windows so we can reuse memory dealers
07         CursorWindow window = new CursorWindow(false /* window will be used remotely */);
08         BulkCursorToCursorAdaptor adaptor = new BulkCursorToCursorAdaptor();
09         IBulkCursor bulkCursor = bulkQueryInternal(
10             url, projection, selection, selectionArgs, sortOrder,
11             adaptor.getObserver(), window,
12             adaptor);
13         if (bulkCursor == null) {
14             return null;
15         }
```

```
16          return adaptor;
17      }
18
19      ......
20  }
```

第7行和第8行代码首先分别创建了一个CursorWindow对象window和一个BulkCursorToCursorAdaptor对象adaptor，接着第9行到第12行代码以这两个对象为参数来调用成员函数bulkQueryInternal，以便可以请求ArticlesProvider组件返回符合条件的博客文章信息。

CursorWindow对象window在创建的过程中，会在内部创建一块匿名共享内存。这块匿名共享内存会与CursorWindow对象window一起传递给ArticlesProvider组件，以便ArticlesProvider组件可以将符合条件的博客文章信息保存在里面。

下面我们首先分析CursorWindow对象window的创建过程，然后分析ContentProviderProxy类的成员函数bulkQueryInternal的实现。

CursorWindow类的构造函数的实现如下所示。

frameworks/base/core/java/android/database/CursorWindow.java

```
01 public class CursorWindow extends SQLiteClosable implements Parcelable {
02      ......
03
04      private int nWindow;
05      ......
06
07      public CursorWindow(boolean localWindow) {
08          ......
09
10          native_init(localWindow);
11      }
12      ......
13
14      /** Does the native side initialization for an empty window */
15      private native void native_init(boolean localOnly);
16
17      ......
18  }
```

Java层中的每一个CursorWindow对象在C++层中都有一个关联的CursorWindow对象。Java层中的CursorWindow对象内部所使用的一块匿名共享内存就是通过C++层中的CursorWindow对象来创建的，因此，在Java层中创建一个CursorWindow对象时，也需要在C++层中创建一个CursorWindow对象，这是通过调用CursorWindow类的成员函数native_init来实现的。

CursorWindow类的成员函数native_init是一个JNI方法，它是由C++层中的函数native_init_empty来实现的。接下来，我们就通过它来分析C++层中的CursorWindow对象的创建过程。

Step 8：CursorWindow.native_init

frameworks/base/core/jni/android_database_CursorWindow.cpp

```
01 static void native_init_empty(JNIEnv * env, jobject object, jboolean localOnly)
02 {
03      ......
04
05      CursorWindow * window;
06
07      window = new CursorWindow(MAX_WINDOW_SIZE);
08      ......
09
10      if (!window->initBuffer(localOnly)) {
11          ......
```

```
12      }
13      ......
14
15      SET_WINDOW(env, object, window);
16  }
```

第7行代码首先使用宏MAX_WINDOW_SIZE来创建一个C++层的CursorWindow对象window，接着第10行代码调用这个CursorWindow对象的成员函数initBuffer在它内部创建一块匿名共享内存，最后第15行代码使用宏SET_WINDOW将前面所创建的CursorWindow对象的地址保存到参数object所描述的一个Java层的CursorWindow对象的成员变量nWindow中。

宏MAX_WINDOW_SIZE的定义如下所示。

frameworks/base/core/jni/CursorWindow.h

```
1 #define MAX_WINDOW_SIZE (1024 * 1024)
```

它的值等于1024 * 1024，表示我们需要在CursorWindow对象window内部创建一块大小为1024 * 1024的匿名共享内存。

CursorWindow类的成员函数initBuffer的实现如下所示。

frameworks/base/core/jni/CursorWindow.cpp

```
01 CursorWindow::CursorWindow(size_t maxSize) :
02      mMaxSize(maxSize)
03 {
04 }
05
06 bool CursorWindow::initBuffer(bool localOnly)
07 {
08      ......
09
10      sp<MemoryHeapBase> heap;
11      heap = new MemoryHeapBase(mMaxSize, 0, "CursorWindow");
12      if (heap != NULL) {
13          mMemory = new MemoryBase(heap, 0, mMaxSize);
14          if (mMemory != NULL) {
15              mData = (uint8_t *) mMemory->pointer();
16              if (mData) {
17                  ......
18                  mSize = mMaxSize;
19                  ......
20              }
21          }
22          ......
23      }
24      ......
25 }
```

CursorWindow类的成员变量mMaxSize是在构造函数中初始化的，它的值就等于MAX_WINDOW_SIZE。第11行代码首先创建了一个MemoryHeapBase对象heap，接着第13行代码将这个MemoryHeapBase对象封装成一个MemoryBase对象，并且保存在CursorWindow类的成员变量mMemory中。从前面6.3.1和6.3.2小节的内容可以知道，MemoryHeapBase类是用来创建一块匿名共享内存的，而MemoryBase类是用来管理使用MemoryHeapBase类创建的一整块匿名共享内存的某一小部分的。最后第15行和第18行代码分别将前面所创建的匿名共享内存的地址和大小保存在CursorWindow类的成员变量mData和mSize中。

宏SET_WINDOW的定义如下所示。

frameworks/base/core/jni/android_database_CursorWindow.cpp

```
1 #define SET_WINDOW(env, object, window) (env->SetIntField(object, gWindowField, (int)window))
```

全局变量gWindowField用来描述Java层中的CursorWindow类的成员变量nWindow在类内部的偏移

量，它是在JNI方法注册函数register_android_database_CursorWindow中初始化的，如下所示。

frameworks/base/core/jni/android_database_CursorWindow.cpp

```
01 static jfieldID gWindowField;
02 ......
03
04 int register_android_database_CursorWindow(JNIEnv * env)
05 {
06     jclass clazz;
07
08     clazz = env->FindClass("android/database/CursorWindow");
09     ......
10
11     gWindowField = env->GetFieldID(clazz, "nWindow", "I");
12
13     ......
14 }
```

因此，宏SET_WINDOW的作用就是将前面所创建的CursorWindow对象window的地址值保存在参数object所描述的一个Java层的CursorWindow对象的成员变量nWindow中，这样就可以将分别在C++层和Java层的这两个CursorWindow对象关联起来。

这一步执行完成之后，返回到前面的Step 7中，即ContentProviderProxy类的成员函数query中，接下来就会调用ContentProviderProxy类的成员函数bulkQueryInternal将前面所创建的匿名共享内存传递给ArticlesProvider组件，以便ArticlesProvider组件可以将与参数url对应的博客文章信息保存在它里面。

Step 9：ContentProviderProxy.bulkQueryInternal

frameworks/base/core/java/android/content/ContentProviderNative.java

```
01 final class ContentProviderProxy implements IContentProvider
02 {
03     ......
04
05     private IBulkCursor bulkQueryInternal(
06             Uri url, String[] projection,
07             String selection, String[] selectionArgs, String sortOrder,
08             IContentObserver observer, CursorWindow window,
09             BulkCursorToCursorAdaptor adaptor) throws RemoteException {
10         Parcel data = Parcel.obtain();
11         Parcel reply = Parcel.obtain();
12
13         data.writeInterfaceToken(IContentProvider.descriptor);
14
15         url.writeToParcel(data, 0);
16         int length = 0;
17         if (projection != null) {
18             length = projection.length;
19         }
20         data.writeInt(length);
21         for (int i = 0; i < length; i++) {
22             data.writeString(projection[i]);
23         }
24         data.writeString(selection);
25         if (selectionArgs != null) {
26             length = selectionArgs.length;
27         } else {
28             length = 0;
29         }
30         data.writeInt(length);
31         for (int i = 0; i < length; i++) {
32             data.writeString(selectionArgs[i]);
33         }
```

```
34          data.writeString(sortOrder);
35          data.writeStrongBinder(observer.asBinder());
36          window.writeToParcel(data, 0);
37
38          // Flag for whether or not we want the number of rows in the
39          // cursor and the position of the "_id" column index (or -1 if
40          // non-existent).  Only to be returned if binder != null.
41          final boolean wantsCursorMetadata = (adaptor != null);
42          data.writeInt(wantsCursorMetadata ? 1 : 0);
43
44          mRemote.transact(IContentProvider.QUERY_TRANSACTION, data, reply, 0);
45
46          DatabaseUtils.readExceptionFromParcel(reply);
47
48          IBulkCursor bulkCursor = null;
49          IBinder bulkCursorBinder = reply.readStrongBinder();
50          if (bulkCursorBinder != null) {
51              bulkCursor = BulkCursorNative.asInterface(bulkCursorBinder);
52
53              if (wantsCursorMetadata) {
54                  int rowCount = reply.readInt();
55                  int idColumnPosition = reply.readInt();
56                  if (bulkCursor != null) {
57                      adaptor.set(bulkCursor, rowCount, idColumnPosition);
58                  }
59              }
60          }
61
62          data.recycle();
63          reply.recycle();
64
65          return bulkCursor;
66      }
67
68      ......
69  }
```

第13行到第42行代码将前面传进来的参数写入到Parcel对象data中，接着再通过ContentProviderProxy类内部的一个Binder代理对象mRemote向ArticlesProvider组件发送一个类型为IContentProvider.QUERY_TRANSACTION的进程间通信请求。

ArticlesProvider组件处理完成这个类型为IContentProvider.QUERY_TRANSACTION的进程间通信请求之后，就会返回一个CursorToBulkCursorAdaptor代理对象bulkCursor给MainActivity组件，里面包含了MainActivity组件所请求的博客文章信息。最后第57行代码将这个CursorToBulkCursorAdaptor代理对象保存在BulkCursorToCursorAdaptor对象adaptor中，以便MainActivity组件可以通过它来获得ArticlesProvider组件所返回的博客文章信息。

在前面传进来的一系列参数中，window和adaptor这两个参数是最重要的，它们分别指向了在前面Step 7中所创建的一个CursorWindow对象和一个BulkCursorToCursorAdaptor对象。第36行代码调用参数window的成员函数writeToParcel将它内部的一块匿名共享内存的信息写入到Parcel对象data中。第41行代码判断参数adaptor是否等于null。如果不等于null，那么第42行代码就会往Parcel对象data中写入一个整数1，表示ArticlesProvider组件在返回符合条件的博客文章信息给MainActivity组件时，需要将这些符合条件的博客文章信息的元数据也一起返回来。

接下来，我们继续分析CursorWindow类的成员函数wrtieToParcel是如何将参数window内部的一块匿名共享内存的信息写入到Parcel对象data中的。

Step 10：CursorWindow.writeToParcel

frameworks/base/core/java/android/database/CursorWindow.java

```
01 public class CursorWindow extends SQLiteClosable implements Parcelable {
02     ......
03
04     public void writeToParcel(Parcel dest, int flags) {
05         ......
06         dest.writeStrongBinder(native_getBinder());
07         ......
08     }
09
10     ......
11 }
```

第6行代码首先调用成员函数native_getBinder来获得一个Binder本地对象，然后将这个Binder本地对象写入到Parcel对象dest中。

CursorWindow类的成员函数native_getBinder是一个JNI方法，它是由C++层中的函数native_getBinder来实现的，下面我们就分析它的实现。

Step 11：CursorWindow.native_getBinder

frameworks/base/core/jni/android_database_CursorWindow.cpp

```
01 static jobject native_getBinder(JNIEnv * env, jobject object)
02 {
03     CursorWindow * window = GET_WINDOW(env, object);
04     if (window) {
05         sp<IMemory> memory = window->getMemory();
06         if (memory != NULL) {
07             sp<IBinder> binder = memory->asBinder();
08             return javaObjectForIBinder(env, binder);
09         }
10     }
11     return NULL;
12 }
```

参数object指向了一个Java层的CursorWindow对象，这个CursorWindow对象是在前面的Step 7中创建的。第3行代码首先获得与它关联的一个C++层的CursorWindow对象window，这是使用宏GET_WINDOW来实现的。

宏GET_WINDOW的定义如下所示。

frameworks/base/core/jni/android_database_CursorWindow.cpp

```
1 #define GET_WINDOW(env, object) ((CursorWindow *)env->GetIntField(object, gWindowField))
```

在前面的Step 8中提到，全局变量gWindowField是用来描述Java层的CursorWindow类的成员变量nWindow在类内部的偏移量的，因此，宏GET_WINDOW实际上就是将Java层中的CursorWindow对象object的成员变量nWindow转换为一个C++层的CursorWindow对象。

返回到函数native_getBinder中，第5行代码接着调用前面获得的CursorWindow对象window的成员函数getMemory来获得它内部的一个MemoryBase对象，最后第8行代码调用函数javaObjectForIBinder将这个MemoryBase对象封装成一个可以在Java层中使用的Binder本地对象，并且将它返回给调用者。

CursorWindow类的成员函数getMemory的实现如下所示。

frameworks/base/core/jni/CursorWindow.h

```
01 class CursorWindow
02 {
03 public:
04     ......
```

```
05      sp<IMemory>          getMemory() {return mMemory;}
06      ......
07
08  private:
09      ......
10      sp<IMemory> mMemory;
11      ......
12  }
```

它将CursorWindow类的成员变量mMemory返回给调用者。从前面的Step 8可以知道，CursorWindow类的成员变量mMemory指向了一个MemeoryBase对象，它是在CursorWindow类的成员函数initBuffer中创建的，用来描述一块匿名共享内存。

这一步执行完成之后，先返回到Step 10中，接着再返回到Step 9中，即ContentProviderProxy类的成员函数bulkQueryInternal中，最后就通过ContentProviderProxy类内部的一个Binder代理对象mRemote向ArticlesProvider组件发送一个类型为IContentProvider.QUERY_TRANSACTION的进程间通信请求。

从前面10.2小节的内容可以知道，ContentProviderProxy类内部的Binder代理对象mRemote实际上引用的是一个Transport对象，这个Transport对象包含在一个Content Provider组件内部，负责接收和处理发送给它的宿主Content Provider组件的进程间通信请求。

以上11步都是在应用程序Article中执行的，接下来第12步到第24步是在应用程序ArticlesProvider中执行的，主要用来处理MainActivity组件发出的类型为IContentProvider.QUERY_TRANSACTION的进程间通信请求。我们首先分析第12步到第14步，如图10-18所示。

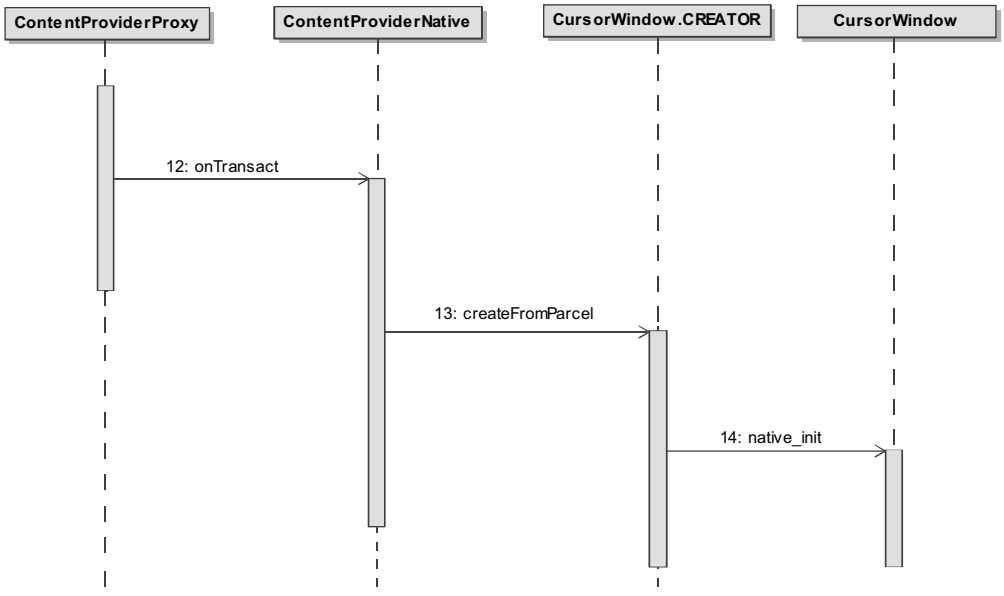


图10-18　ArticlesProvider组件传输数据的过程——Step 12至Step 14

Step 12: ContentProviderNative.onTransact

frameworks/base/core/java/android/content/ContentProviderNative.java

```
01 abstract public class ContentProviderNative extends Binder implements IContentProvider {
02      ......
03
04      @Override
05      public boolean onTransact(int code, Parcel data, Parcel reply, int flags)
06      throws RemoteException {
07          try {
08              switch (code) {
09              case QUERY_TRANSACTION:
```

```
10                 {
11                     data.enforceInterface(IContentProvider.descriptor);
12
13                     Uri url = Uri.CREATOR.createFromParcel(data);
14
15                     // String[] projection
16                     int num = data.readInt();
17                     String[] projection = null;
18                     if (num > 0) {
19                         projection = new String[num];
20                         for (int i = 0; i < num; i++) {
21                             projection[i] = data.readString();
22                         }
23                     }
24
25                     // String selection, String[] selectionArgs...
26                     String selection = data.readString();
27                     num = data.readInt();
28                     String[] selectionArgs = null;
29                     if (num > 0) {
30                         selectionArgs = new String[num];
31                         for (int i = 0; i < num; i++) {
32                             selectionArgs[i] = data.readString();
33                         }
34                     }
35
36                     String sortOrder = data.readString();
37                     IContentObserver observer = IContentObserver.Stub.
38                         asInterface(data.readStrongBinder());
39                     CursorWindow window = CursorWindow.CREATOR.createFromParcel(data);
40
41                     // Flag for whether caller wants the number of
42                     // rows in the cursor and the position of the
43                     // "_id" column index (or -1 if non-existent)
44                     // Only to be returned if binder != null.
45                     boolean wantsCursorMetadata = data.readInt() != 0;
46
47                     IBulkCursor bulkCursor = bulkQuery(url, projection, selection,
48                         selectionArgs, sortOrder, observer, window);
49                     reply.writeNoException();
50                     if (bulkCursor != null) {
51                         reply.writeStrongBinder(bulkCursor.asBinder());
52
53                         if (wantsCursorMetadata) {
54                             reply.writeInt(bulkCursor.count());
55                             reply.writeInt(BulkCursorToCursorAdaptor.findRowIdColumnIndex(
56                                 bulkCursor.getColumnNames()));
57                         }
58                     } else {
59                         reply.writeStrongBinder(null);
60                     }
61
62                     return true;
63                 }
64             ......
65             }
66         } catch (Exception e) {
67             DatabaseUtils.writeExceptionToParcel(reply, e);
68             return true;
69         }
70
71         return super.onTransact(code, data, reply, flags);
72     }
73
74     ......
75 }
```

ContentProviderNative类的成员函数onTransact负责处理类型为IContentProvider.QUERY_TRANSACTION的进程间通信请求。

第11行到第45行代码将Parcel对象data中的进程间通信数据读取出来，这些进程间通信数据是从前面的Step 9中传递过来的。在前面的Step 9中提到，在这些进程间通信数据中，最后两个进程间通信数据是最重要的，其中，倒数第二个进程间通信数据是一个类型为MemoryBase的Binder本地对象，用来描述一块匿名共享内存；而倒数第一个进程间通信数据是一个整数，用来描述ArticlesProvider组件是否需要返回符合条件的博客文章信息的元数据。第39行代码调用CursorWindow类的一个静态成员变量CREATOR的成员函数createFromParcel将倒数第二个进程间通信数据封装成一个CursorWindow对象window。第45行代码将倒数第一个进程间通数据所描述的一个整数读取出来。如果它的值不等于0，那么就将布尔变量wantsCursorMetadata的值设置为true，表示ArticlesProvider组件需要将符合条件的博客文章信息的元数据返回给MainActivity组件。

第47行和第48行代码调用Transport类的成员函数bulkQuery来获得一个CursorToBulkCursorAdaptor对象bulkCursor。这个CursorToBulkCursorAdaptor对象包含了ArticlesProvider组件返回给MainActivity组件的博客文章信息。接着第51行代码将它写入到Parcel对象reply中，以便ArticlesProvider组件可以将它返回给MainActivity组件。

从前面的调用过程可以知道，Parcel对象data的最后一个进程间通数据的值等于1，即前面得到的布尔变量wantsCursorMetadata的值等于true。因此，接下来第54行到第56行代码就会获得保存在CursorToBulkCursorAdaptor对象bulkCursor中的博客文章信息的元数据，并且将它们写入到Parcel对象reply中去返回给MainActivity组件。

第54行代码调用CursorToBulkCursorAdaptor对象bulkCursor的成员函数count来获得需要返回给MainActivity组件的博客文章条目的总数；而第55行和第56行代码首先调用CursorToBulkCursorAdaptor对象bulkCursor的成员函数getColumnNames来获得用来描述一个博客文章条目的字段名称集合，然后调用BulkCursorToCursorAdaptor类的静态成员函数findRowIdColumnIndex在这个集合中找到一个名称为“_id”的字段，最后就可以获得它在集合中的位置值了。从前面10.1.1小节的内容可以知道，用来保存博客文章条目的数据库表的主键的字段名称被设置为“_id”，它是用来描述一个博客文章条目的ID值的。MainActivity组件获得了这个字段的位置值之后，就可以方便地使用CursorAdapter接口来访问ArticlesProvider组件返回给它的博客文章信息了。

接下来，我们首先分析CursorWindow类的静态成员变量CREATOR的成员函数createFromParcel是如何将倒数第二个进程间通信数据封装成一个CursorWindow对象的，然后分析Transport类的成员函数bulkQuery是如何返回一个包含了MainActivity组件所请求的博客文章信息的CursorToBulkCursorAdaptor对象的，最后分析前面所获得的CursorToBulkCursorAdaptor对象的成员函数count是如何获得保存在它里面的博客文章条目的总数的。

Step 13：CursorWindow.CREATOR.createFromParcel

frameworks/base/core/java/android/database/CursorWindow.java

```
01 public class CursorWindow extends SQLiteClosable implements Parcelable {
02     ......
03
04     public static final Parcelable.Creator<CursorWindow> CREATOR
05             = new Parcelable.Creator<CursorWindow>() {
06         public CursorWindow createFromParcel(Parcel source) {
07             return new CursorWindow(source);
08         }
```

```
09
10          ......
11      };
12
13      ......
14  }
```

第7行代码使用参数source来创建一个CursorWindow对象，创建过程如下所示。

frameworks/base/core/java/android/database/CursorWindow.java

```
01 public class CursorWindow extends SQLiteClosable implements Parcelable {
02      ......
03
04      private CursorWindow(Parcel source) {
05          IBinder nativeBinder = source.readStrongBinder();
06          ......
07
08          native_init(nativeBinder);
09      }
10      ......
11
12      private native void native_init(IBinder nativeBinder);
13
14      ......
15 }
```

从前面的调用过程可以知道，参数source的当前位置是一个类型为MemoryBase的Binder代理对象。因此，第5行代码就可以将它从参数source中读取出来，接着第8行代码再以这个Binder代理对象为参数来调用CursorWindow类的成员函数native_init，以便可以在C++层中创建一个关联的CursorWindow对象。

CursorWindow类的成员函数native_init是一个JNI方法，它是由C++层中的函数native_init_memory来实现的。下面我们就通过它来分析C++层中的CursorWindow对象的创建过程。

Step 14：CursorWindow.native_init

frameworks/base/core/jni/android_database_CursorWindow.cpp

```
01 static void native_init_memory(JNIEnv * env, jobject object, jobject memObj)
02 {
03      sp<IMemory> memory = interface_cast<IMemory>(ibinderForJavaObject(env, memObj));
04      ......
05
06      CursorWindow * window = new CursorWindow();
07      ......
08      if (!window->setMemory(memory)) {
09          ......
10      }
11
12      ......
13      SET_WINDOW(env, object, window);
14 }
```

参数memObj指向了一个Java层的Binder代理对象，这个Binder代理对象引用了一个类型为MemoryBase的Binder本地对象。因此，第3行代码就调用函数ibinderForJavaObject将它转换为一个C++层的类型为MemoryBase的Binder代理对象memory。

第6行代码创建了一个C++层的CursorWindow对象，接着第8行代码调用它的成员函数setMemory来设置它内部所引用的一块匿名共享内存。

参数object指向了一个Java层的CursorWindow对象，它是在前面的Step 13中创建的，第13行代码

使用宏SET_WINDOW将前面所创建的CursorWindow对象window的地址值保存在它的成员变量nWindow中，这样就可以将分别在Java层和C++层的这两个CursorWindow对象关联起来。

CursorWindow类的成员函数setMemory的实现如下所示。

frameworks/base/core/jni/CursorWindow.cpp

```
01 bool CursorWindow::setMemory(const sp<IMemory>& memory)
02 {
03     mMemory = memory;
04     mData = (uint8_t *) memory->pointer();
05     ......
06
07     // Make the window read-only
08     ssize_t size = memory->size();
09     mSize = size;
10     ......
11
12     return true;
13 }
```

参数memory指向了一个MemoryBase代理对象。第3行代码首先将它保存在成员变量mMemory中，接着第4行和第8行代码分别调用它的成员函数pointer和size来获得它内部所引用的一块匿名共享内存在当前应用程序进程中的地址以及大小，并且分别保存在成员变量mData和mSize中。

这一步执行完成之后，返回到前面的Step 12中，即ContentProviderNative类的成员函数onTransact中，我们就获得了一个CursorWindow对象，这个CursorWindow对象与在前面Step 7中创建的CursorWindow对象共享同一块匿名共享内存。接下来第15步到第20步就以这个CursorWindow对象为参数来调用Transport类的成员函数bulkQuery，以便可以从ArticlesProvider组件中获得MainActivity组件所请求的博客文章信息，如图10-19所示。

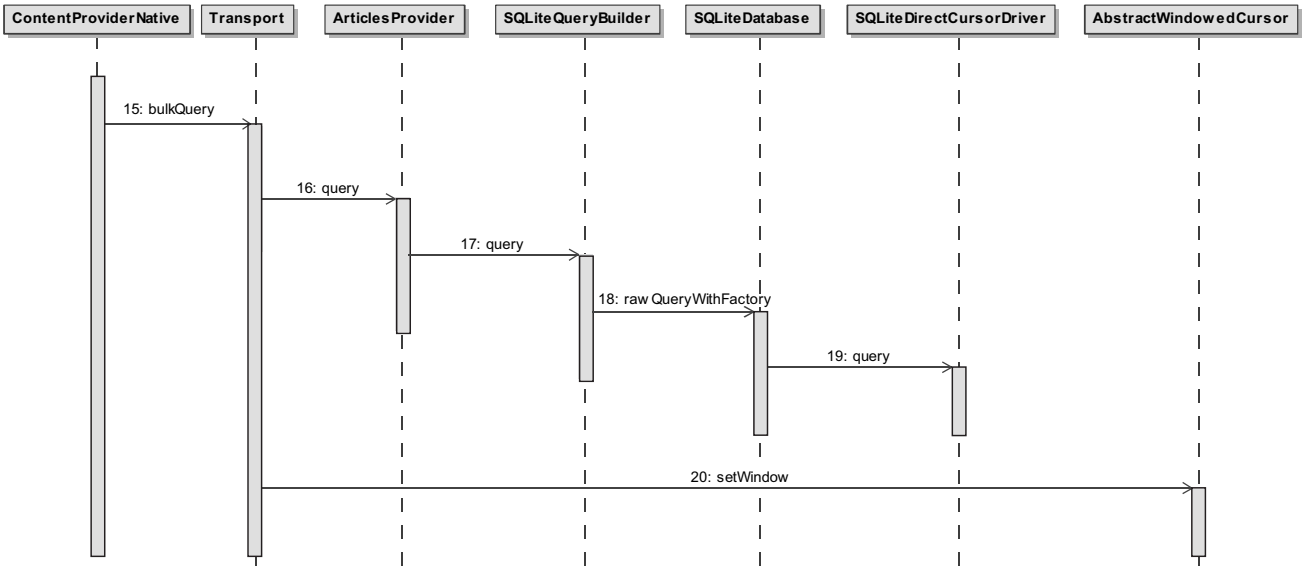


图10-19　ArticlesProvider组件传输数据的过程——Step 15至Step 20

Step 15：Transport.bulkQuery

frameworks/base/core/java/android/content/ContentProvider.java

```
01 public abstract class ContentProvider implements ComponentCallbacks {
02     ......
03
04     class Transport extends ContentProviderNative {
05         ......
```

```
06
07          public IBulkCursor bulkQuery(Uri uri, String[] projection,
08                  String selection, String[] selectionArgs, String sortOrder,
09                  IContentObserver observer, CursorWindow window) {
10              ......
11
12              Cursor cursor = ContentProvider.this.query(uri, projection,
13                  selection, selectionArgs, sortOrder);
14              ......
15
16              return new CursorToBulkCursorAdaptor(cursor, observer,
17                  ContentProvider.this.getClass().getName(),
18                  hasWritePermission(uri), window);
19          }
20
21          ......
22      }
23
24      ......
25  }
```

第12行和第13行代码调用ContentProvider类的成员函数query来获得MainActivity组件所请求的博客文章信息，并且将它们保存在一个Cursor对象cursor中。第16行到第18行代码首先将Cursor对象cursor和CursorWindow对象window封装成一个CursorToBulkCursorAdaptor对象，然后将这个CursorToBulkCursorAdaptor对象返回给调用者。

Cursor对象cursor的实际类型为SQLiteCursor，保存在它里面的博客文章信息是使用一个数据库查询计划来描述的。这个数据库查询计划要等到它所描述的博客文章信息第一次被访问时才会被执行，执行结果就保存在CursorWindow对象window内部的一块匿名共享内存中。

ContentProvider类的成员函数query是由其子类实现的。在我们这个场景中，这个子类就是ArticlesProvider组件。接下来，我们首先分析ArticlesProvider组件的成员函数query的实现，然后分析一个CursorToBulkCursorAdaptor对象的创建过程。

Step 16：ArticlesProvider.query

packages/experimental/ArticlesProvider/src/shy/luo/providers/articles/ArticlesProvider.java

```
01 public class ArticlesProvider extends ContentProvider {
02      ......
03
04      @Override
05      public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) {
06          SQLiteDatabase db = dbHelper.getReadableDatabase();
07
08          SQLiteQueryBuilder sqlBuilder = new SQLiteQueryBuilder();
09          String limit = null;
10
11          switch (uriMatcher.match(uri)) {
12              ......
13              case Articles.ITEM_POS: {
14                  String pos = uri.getPathSegments().get(1);
15                  sqlBuilder.setTables(DB_TABLE);
16                  sqlBuilder.setProjectionMap(articleProjectionMap);
17                  limit = pos + ", 1";
18                  break;
19              }
20              ......
21          }
22
23            Cursor cursor = sqlBuilder.query(db, projection, selection, selectionArgs, null,
null, TextUtils.isEmpty(sortOrder) ? Articles.DEFAULT_SORT_ORDER : sortOrder, limit);
```

```
24          ......
25
26          return cursor;
27      }
28
29      ......
30  }
```

从前面的Step 1可以知道，参数uri的值为“content://shy.luo.providers.articles/pos/#”，其中，符号“#”是一个数字，用来描述MainActivity组件所请求的博客文章条目在ArticlesProvider组件中的位置，即在数据库表中的位置。第14行到第17行代码根据这个位置构造了一条SQL查询语句，保存在一个SQLiteQueryBuilder对象sqlBuilder中。

ArticlesProvider组件中的博客文章信息是保存在SQLiteDatabase对象db所描述的一个SQLite数据库中的，因此，第23行代码就以它为参数来调用SQLiteQueryBuilder对象sqlBuilder的成员函数query，以便可以创建一个数据库查询计划。这个数据库查询计划保存在一个SQLiteCursor对象中，用来获取MainActivity组件所请求的博客文章条目。

Step 17：SQLiteQueryBuilder.query

frameworks/base/core/java/android/database/sqlite/SQLiteQueryBuilder.java

```
01 public class SQLiteQueryBuilder
02 {
03      ......
04
05      public Cursor query(SQLiteDatabase db, String[] projectionIn,
06              String selection, String[] selectionArgs, String groupBy,
07              String having, String sortOrder, String limit) {
08          ......
09
10          String sql = buildQuery(
11              projectionIn, selection, groupBy, having,
12              sortOrder, limit);
13          ......
14
15          return db.rawQueryWithFactory(
16              mFactory, sql, selectionArgs,
17              SQLiteDatabase.findEditTable(mTables));
18      }
19
20      ......
21 }
```

第10行到第12行代码首先调用成员函数buildQuery将参数projectionIn、selection、groupBy、having、sortOrder和limit封装成一条SQL查询语句，接着第15行到第17行代码调用SQLiteDatabase对象db的成员函数rawQueryWithFactory将这条SQL查询语句封装成一个数据库查询计划，并且保存在一个SQLiteCursor对象中。

Step 18：SQLiteDatabase.rawQueryWithFactory

frameworks/base/core/java/android/database/sqlite/SQLiteDatabase.java

```
01 public class SQLiteDatabase extends SQLiteClosable {
02      ......
03
04      public Cursor rawQueryWithFactory(
05              CursorFactory cursorFactory, String sql, String[] selectionArgs,
06              String editTable) {
07          ......
08
09          SQLiteCursorDriver driver = new SQLiteDirectCursorDriver(this, sql, editTable);
```

```
10
11          Cursor cursor = null;
12          try {
13              cursor = driver.query(
14                  cursorFactory != null ? cursorFactory : mFactory,
15                  selectionArgs);
16          } finally {
17              ......
18          }
19
20          return cursor;
21      }
22
23      ......
24  }
```

第9行代码首先将参数sql所描述的一条SQL查询语句封装在一个SQLiteDirectCursorDriver对象中，接着第13行到第15行代码调用这个SQLiteDirectCursorDriver对象的成员函数query来创建一个SQLiteCursor对象，最后第20行代码将这个SQLiteCursor对象返回给调用者。

Step 19：SQLiteDirectCursorDriver.query

frameworks/base/core/java/android/database/sqlite/SQLiteDirectCursorDriver.java

```
01 public class SQLiteDirectCursorDriver implements SQLiteCursorDriver {
02      ......
03
04      public Cursor query(CursorFactory factory, String[] selectionArgs) {
05          // Compile the query
06          SQLiteQuery query = new SQLiteQuery(mDatabase, mSql, 0, selectionArgs);
07
08          try {
09              ......
10
11              // Create the cursor
12              if (factory == null) {
13                  mCursor = new SQLiteCursor(mDatabase, this, mEditTable, query);
14              } else {
15                  mCursor = factory.newCursor(mDatabase, this, mEditTable, query);
16              }
17
18              ......
19              return mCursor;
20          } finally {
21              ......
22          }
23      }
24
25      ......
26  }
```

第6行代码首先根据SQLiteDirectCursorDriver类的成员变量mDatabase和mSql创建了一个数据库查询计划，保存在一个SQLiteQuery对象query中。接着第12行到第16行代码将这个数据库查询计划保存在一个SQLiteCursor对象中。最后第19行代码将这个SQLiteCursor对象返回给调用者。

这一步执行完成之后，就沿着调用路径返回到前面的Step 15中，即Transport类的成员函数bulkQuery中，接下来就会调用CursorToBulkCursorAdaptor类的构造函数将这一步所创建的一个SQLiteCursor对象，以及参数window所描述的一个CursorWindow对象封装成一个CursorToBulkCursorAdaptor对象。

CursorToBulkCursorAdaptor类的构造函数的实现如下所示。

frameworks/base/core/java/android/database/CursorToBulkCursorAdaptor.java

```
01 public final class CursorToBulkCursorAdaptor extends BulkCursorNative
02         implements IBinder.DeathRecipient {
03     ......
04
05     public CursorToBulkCursorAdaptor(Cursor cursor, IContentObserver observer, String providerName,
06             boolean allowWrite, CursorWindow window) {
07         try {
08             mCursor = (CrossProcessCursor) cursor;
09             if (mCursor instanceof AbstractWindowedCursor) {
10                 AbstractWindowedCursor windowedCursor = (AbstractWindowedCursor) cursor;
11                 ......
12
13                 windowedCursor.setWindow(window);
14             } else {
15                 ......
16             }
17         } catch (ClassCastException e) {
18             ......
19         }
20
21         ......
22     }
23
24     ......
25 }
```

参数cursor指向了一个SQLiteCursor对象，它实现了CrossProcessCursor接口，并且继承了AbstractWindowedCursor类，因此，第9行的if语句为true。接下来第10行代码首先将参数cursor转换成一个AbstractWindowedCursor对象，然后第13行代码再调用它的成员函数setWindow将参数window所描述的一个CursorWindow对象保存在它里面。

Step 20：AbstractWindowedCursor.setWindow

frameworks/base/core/java/android/database/AbstractWindowedCursor.java

```
01 public abstract class AbstractWindowedCursor extends AbstractCursor
02 {
03     ......
04
05     public void setWindow(CursorWindow window) {
06         ......
07
08         mWindow = window;
09     }
10     ......
11
12     protected CursorWindow mWindow;
13 }
```

第8行代码将参数window所描述的一个CursorWindow对象保存到AbstractWindowedCursor类的成员变量mWindow中。

注意

由于AbstractWindowedCursor类的成员变量mWindow的访问权限为protected，因此，AbstractWindowedCursor的子类可以直接访问它。

这一步执行完成之后，沿着调用路径返回到前面的Step 12中，即ContentProviderNative类的成员函数onTransact函数中，接下来第21步到第24步就会获得MainActivity组件所请求的博客文章条目的总数，这是通过调用在前面Step 15中所创建的一个CursorToBulkCursorAdaptor对象的成员函数count来实现的，如图10–20所示。

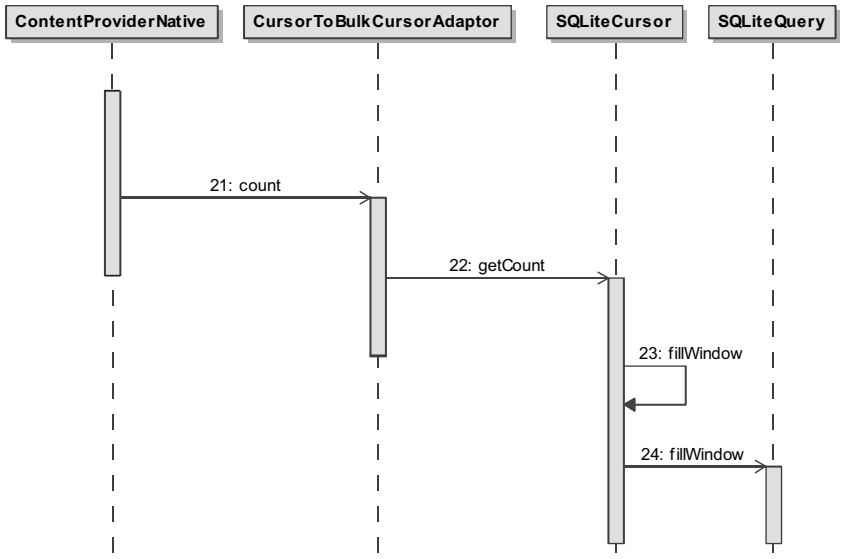


图10-20　ArticlesProvider组件传输数据的过程——Step 21至Step 24

Step 21：CursorToBulkCursorAdaptor.count

frameworks/base/core/java/android/database/CursorToBulkCursorAdaptor.java

```
01 public final class CursorToBulkCursorAdaptor extends BulkCursorNative
02         implements IBinder.DeathRecipient {
03     ......
04     private final CrossProcessCursor mCursor;
05     ......
06
07     public int count() {
08         return mCursor.getCount();
09     }
10
11     ......
12 }
```

CursorToBulkCursorAdaptor类的成员变量mCursor指向了一个SQLiteCursor对象，这个SQLiteCursor对象是在前面的Step 19中创建的，因此，接下来就会调用它的成员函数getCount来获得MainActivity组件所请求的博客文章条目的总数。

Step 22：SQLiteCursor.getCount

frameworks/base/core/java/android/database/sqlite/SQLiteCursor.java

```
01 public class SQLiteCursor extends AbstractWindowedCursor {
02     ......
03
04     /** The number of rows in the cursor */
05     private int mCount = NO_COUNT;
06     ......
07
08     @Override
09     public int getCount() {
10         if (mCount == NO_COUNT) {
11             fillWindow(0);
12         }
13         return mCount;
14     }
15
16     ......
17 }
```

SQLiteCursor类的成员变量mCount的初始值为NO_COUNT，表示包含在SQLiteCursor类中的数据库查询计划还没有被执行。当这个数据库查询计划被执行之后，SQLiteCursor类的成员变量mCount就用来保存从数据库中获得的数据的总行数。在我们这个场景中，这个数字就代表了MainActivity组件所请求的博客文章条目的总数。

第11行代码调用了SQLiteCursor类的成员函数fillWindow来执行它内部的数据库查询计划，以便可以获得MainActivity组件所请求的博客文章条目的总数。

Step 23：SQLiteCursor.fillWindow

frameworks/base/core/java/android/database/sqlite/SQLiteCursor.java

```
01 public class SQLiteCursor extends AbstractWindowedCursor {
02     ......
03
04     /** The query object for the cursor */
05     private SQLiteQuery mQuery;
06     ......
07
08     private void fillWindow (int startPos) {
09         ......
10
11         mCount = mQuery.fillWindow(mWindow, mInitialRead, 0);
12
13         ......
14     }
15
16     ......
17 }
```

SQLiteCursor类内部所包含的数据库查询计划是通过成员变量mQuery来描述的。第11行代码调用它的成员函数fillWindow来执行它所描述的数据库查询计划，以便可以获得MainActivity组件所请求的博客文章条目的总数。

> **注意**
>
> SQLiteCursor类的成员变量mWindow是从父类AbstractWindowedCursor继承下来的，它里面引用了一块匿名共享内存。当SQLiteCursor类的成员变量mQuery所描述的数据库查询计划执行完成之后，从数据库中获得的数据，即MainActivity组件所请求的博客文章条目就会保存在这块匿名共享内存中。

Step 24：SQLiteQuery.fillWindow

frameworks/base/core/java/android/database/sqlite/SQLiteQuery.java

```
01 public class SQLiteQuery extends SQLiteProgram {
02     ......
03
04     /* package */ int fillWindow(CursorWindow window,
05             int maxRead, int lastPos) {
06         ......
07         try {
08             ......
09             try {
10                 ......
11                 // if the start pos is not equal to 0, then most likely window is
12                 // too small for the data set, loading by another thread
13                 // is not safe in this situation. the native code will ignore maxRead
14                 int numRows = native_fill_window(window, window.getStartPosition(), mOffsetIndex,
15                     maxRead, lastPos);
```

```
16
17                  ......
18                  return numRows;
19              } catch (IllegalStateException e){
20                  ......
21              } catch (SQLiteDatabaseCorruptException e) {
22                  ......
23              } finally {
24                  ......
25              }
26          } finally {
27              ......
28          }
29      }
30
31      ......
32  }
```

第14行和第15行代码调用了SQLiteQuery类的成员函数native_fill_window来执行在前面的Step 19中所创建的一个数据库查询计划，执行结果就保存在参数window所描述的一个CursorWindow对象的内部所引用的一块匿名共享内存中。SQLiteQuery类的成员函数native_fill_window是一个JNI方法，它的返回值numRows表示从数据库中获得的数据的总行数，即MainActivity组件所请求的博客文章条目的总数。

这一步执行完成之后，沿着调用路径返回到前面的Step 12中，即ContentProviderNative类的成员函数onTransact函数中，这时候MainActivity组件所请求的博客文章条目就保存在一个CursorToBulkCursorAdaptor对象中了，因此，最后ArticlesProvider组件就将它返回给MainActivity组件。

至此，ArticlesProvider组件将它里面的博客文章信息传输给MainActivity组件访问的过程就分析完成了。结合前面10.3.1小节所介绍的数据共享模型，我们就可以更加深入地理解Content Provider组件在不同的应用程序之间进行数据共享的原理了。

由于Content Provider组件中的数据是可以在不同的应用程序之间进行共享的，因此，这里就会存在一个数据一致性问题，即当一个应用程序更新Content Provider组件中的一个数据时，如何保证其他应用程序可以看到这个数据更新，以便保证所有的应用程序获得的共享数据都是最新的，并且是一致的。接下来，我们就带着这个问题来分析Content Provider组件的数据更新通知机制。

10.4 Content Provider组件的数据更新通知机制

Content Provider组件的数据更新通知机制类似于Android系统的广播机制，它们都是一种消息发布和订阅的事件驱动模型。在Content Provider组件的数据更新通知机制中，内容观察者（Content Observer）负责接收数据更新通知，而Content Provider组件负责发送数据更新通知。

内容观察者在接收数据更新通知之前，必须要先注册到ContentService中，并且告诉ContentService它要接收什么样的数据更新通知。这是通过一个URI来描述的。当Content Provider组件中的数据发生更新时，Content Provider组件就会将用来描述这个数据的一个URI发送给ContentService，以便ContentService可以找到与这个URI对应的内容观察者，最后向它们发送一个数据更新通知。

接下来，我们就以前面10.1小节中的应用程序为例，来分析Content Provider组件的数据更新通知机制。其中，10.4.1小节分析内容观察者的注册过程，10.4.2小节分析Content Provider组件发送数据更新通知的过程。

10.4.1　注册内容观察者

在前面的10.1.2小节中，MainActivity组件在启动时，会将在内部实现的一个内容观察者注册到ContentService中，用来监控ArticlesProvider组件中的博客文章信息的变化。本节就结合这个使用场景，来分析内容观察者的注册过程。

MainActivity组件在内部实现的一个内容观察者的类型为ArticleObserver，它是从ContentObserver类继承下来的，它的注册过程如图10-21所示。

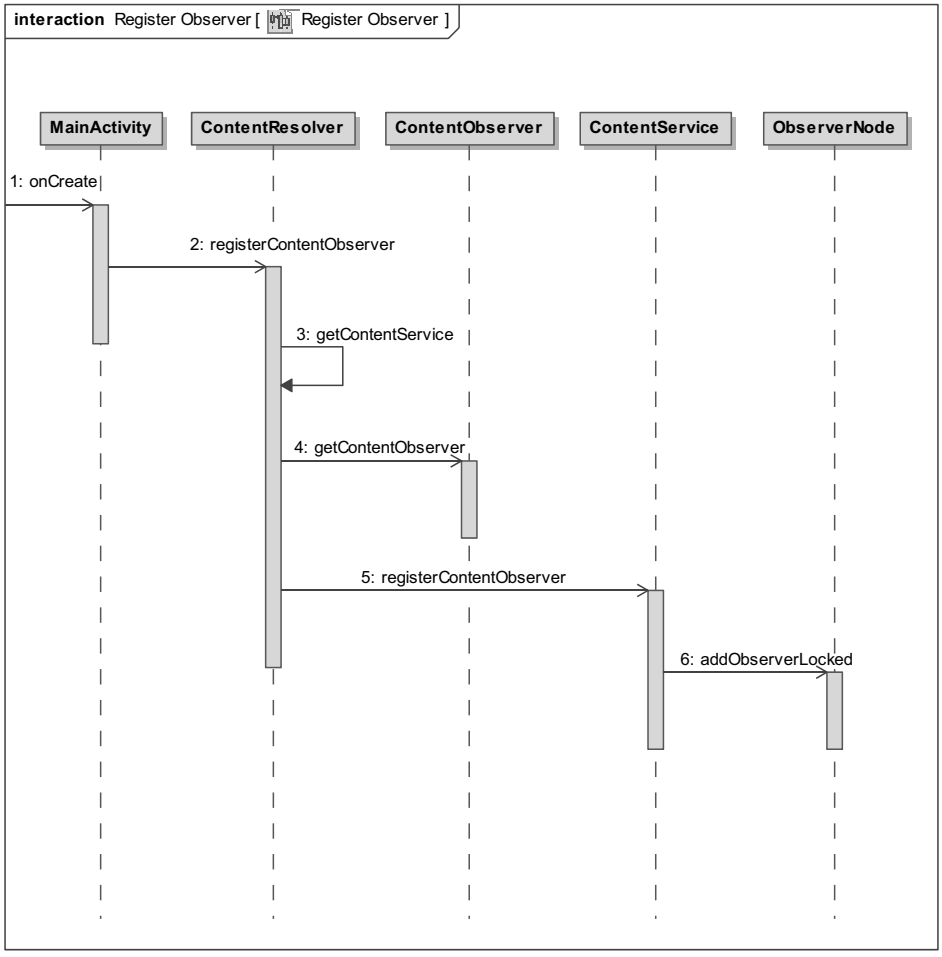


图10-21　内容观察者的注册过程

这个过程一共分为6个步骤，接下来我们就详细分析每一个步骤。

Step 1：MainActivity.onCreate

packages/experimental/Article/src/shy/luo/article/MainActivity.java

```
01 public class MainActivity extends Activity implements View.OnClickListener, AdapterView.OnItemClickListener {
02     ......
03
04     private ArticleAdapter adapter = null;
05     private ArticleObserver observer = null;
06     ......
07
08     @Override
09     public void onCreate(Bundle savedInstanceState) {
10         super.onCreate(savedInstanceState);
```

```
        ......

        observer = new ArticleObserver(new Handler());
        getContentResolver().registerContentObserver(Articles.CONTENT_URI, true, observer);

        ......
    }
    ......

    private class ArticleObserver extends ContentObserver {
        public ArticleObserver(Handler handler) {
            super(handler);
        }

        @Override
        public void onChange (boolean selfChange) {
            adapter.notifyDataSetChanged();
        }
    }

    ......
}
```

自定义的内容观察者必须要继承ContentObserver类，并且需要重写它的成员函数onChange。当一个内容观察者所监控的数据发生更新时，它的成员函数onChange就会被调用，其中，参数selfChange表示这个数据更新是否是由它自己触发的。MainActivity组件在内部实现了一个类型为ArticleObserver的内容观察者，用来监控ArticlesProvider组件中的博客文章信息的变化。

MainActivity组件在启动过程中，即当它的成员函数onCreate被调用时，首先会创建一个类型为ArticleObserver的内容观察者observer，然后调用成员函数getContentResolver来获得一个ContentResolver对象，最后调用这个ContentResolver对象的成员函数registerContentObserver将前面所创建的内容观察者observer注册到ContentService中，并且指定它所要监控的数据是由Articles.CONTENT_URI这个URI来描述的。Articles.CONTENT_URI的值等于“content://shy.luo.providers.articles/item”，它所描述的数据就是ArticlesProvider组件的博客文章信息。

注意

MainActivity组件在创建内容观察者observer时，指定了一个Handler对象，这个Handler对象是用来向MainActivity组件所运行在的应用程序进程的主线程的消息队列发送消息的。这个Handler对象是在接收数据更新通知时使用的，在接下来的10.4.2小节中，我们就会看到它的作用。

Step 2：ContentResolver.registerContentObserver

frameworks/base/core/java/android/content/ContentResolver.java

```
public abstract class ContentResolver {
    ......

    public final void registerContentObserver(Uri uri, boolean notifyForDescendents,
        ContentObserver observer)
    {
        try {
            getContentService().registerContentObserver(uri, notifyForDescendents,
                observer.getContentObserver());
        } catch (RemoteException e) {
        }
    }

    ......
}
```

前面传进来的参数notifyForDescendents的值为true，表示参数observer所描述的内容观察者除了需要监控参数uri所描述的URI所对应的数据更新之外，还需要监控所有以参数uri为前缀的URI所对应的数据更新。

第8行和第9行代码首先调用内容观察者observer的成员函数getContentObserver来获得一个Binder本地对象，然后调用ContentResolver类的成员函数getContentService来获得一个ContentService代理对象，最后调用这个ContentService代理对象的成员函数registerContentObserver将前面所获得的一个Binder本地对象注册到ContentService中。从这里就可以看出，真正注册到ContentService中的并不是一个内容观察者，而是与这个内容观察者所关联的一个Binder本地对象，这个Binder本地对象将代替它所关联的内容观察者来接收ContentService发送过来的数据更新通知。

接下来，我们首先分析ContentService代理对象的获取过程，然后分析ContentObserver类的成员函数getContentObserver的实现。

Step 3：ContentResolver.getContentService

frameworks/base/core/java/android/content/ContentResolver.java

```
01 public abstract class ContentResolver {
02     ......
03
04     public static IContentService getContentService() {
05         if (sContentService != null) {
06             return sContentService;
07         }
08         IBinder b = ServiceManager.getService(CONTENT_SERVICE_NAME);
09         ......
10         sContentService = IContentService.Stub.asInterface(b);
11         ......
12         return sContentService;
13     }
14
15     private static IContentService sContentService;
16     ......
17 }
```

ContentResolver类有一个类型为IContentService的静态成员变量sContentService，它的初始值为null。当ContentResolver类的成员函数getContentService第一次被调用时，ContentResolver类便会到Service Manager中去获得ContentService的一个代理对象[1]，并且将它保存在ContentResolver类的静态成员变量sContentService中。这样，当ContentResolver类的成员函数getContentService再次被调用时，它就可以直接将静态成员变量sContentService所描述的ContentService代理对象返回给调用者了。

Step 4：ContentObserver.getContentObserver

frameworks/base/core/java/android/database/ContentObserver.java

```
01 public abstract class ContentObserver {
02     ......
03
04     private Transport mTransport;
05     ......
06
07     private static final class Transport extends IContentObserver.Stub {
08         ContentObserver mContentObserver;
09
10         public Transport(ContentObserver contentObserver) {
```

1　在接下来的第11章中介绍系统进程System的启动过程时，我们就会看到ContentService是在系统进程System中启动的，并且它注册到Service Manager中的名称为CONTENT_SERVICE_NAME。

```
11              mContentObserver = contentObserver;
12          }
13
14          ......
15      }
16      ......
17
18      public IContentObserver getContentObserver() {
19          synchronized(lock) {
20              if (mTransport == null) {
21                  mTransport = new Transport(this);
22              }
23              return mTransport;
24          }
25      }
26
27      ......
28  }
```

ContentObserver类内部有一个成员变量mTransport，它指向了一个类型为Transport的Binder本地对象，可以通过调用ContentObserver类的成员函数getContentObserver来获得。

注意

这个类型为Transport的Binder本地对象内部有一个类型为ContentObserver的成员变量mContentObserver，它指向了一个外部的内容观察者，当它接收到ContentService发送过来的一个数据更新通知时，便会将这个数据更新通知转发给这个外部的内容观察者处理。

回到前面的Step 2中，即ContentResolver类的成员函数registerContentObserver中，接下来就会调用ContentService代理对象的成员函数registerContentObserver向ContentService发送一个进程间通信请求，以便可以将前面所获得的一个类型为Transport的Binder本地对象注册到它里面。

ContentService接收到这个注册内容观察者的进程间通信请求之后，就会调用它的成员函数registerContentObserver来进一步处理。

Step 5：ContentService.registerContentObserver

frameworks/base/core/java/android/content/ContentService.java

```
01 public final class ContentService extends IContentService.Stub {
02      ......
03
04      private final ObserverNode mRootNode = new ObserverNode("");
05      ......
06
07      public void registerContentObserver(Uri uri, boolean notifyForDescendents,
08              IContentObserver observer) {
09          ......
10
11          synchronized (mRootNode) {
12              mRootNode.addObserverLocked(uri, observer, notifyForDescendents, mRootNode);
13              ......
14          }
15      }
16
17      ......
18  }
```

ContentService在内部使用一棵树来维护所有注册到它里面的内容观察者，这棵树的根节点是通过成员变量mRootNode来描述的，第12行代码调用它的成员函数addObserverLocked将即将要注册的内容观察者observer添加到它里面去。

ContentService类的成员变量mRootNode的类型为ObserverNode，接下来我们就分析它的成员函数addObserverLocked的实现。

Step 6：ObserverNode.addObserverLocked

frameworks/base/core/java/android/content/ContentService.java

```
01 public final class ContentService extends IContentService.Stub {
02     ......
03
04     public static final class ObserverNode {
05         ......
06
07         private String mName;
08         private ArrayList<ObserverNode> mChildren = new ArrayList<ObserverNode>();
09         private ArrayList<ObserverEntry> mObservers = new ArrayList<ObserverEntry>();
10
11         public ObserverNode(String name) {
12             mName = name;
13         }
14         ......
15
16         public void addObserverLocked(Uri uri, IContentObserver observer,
17                 boolean notifyForDescendents, Object observersLock) {
18             addObserverLocked(uri, 0, observer, notifyForDescendents, observersLock);
19         }
20
21         private void addObserverLocked(Uri uri, int index, IContentObserver observer,
22                 boolean notifyForDescendents, Object observersLock) {
23             // If this is the leaf node add the observer
24             if (index == countUriSegments(uri)) {
25                 mObservers.add(new ObserverEntry(observer, notifyForDescendents, observersLock));
26                 return;
27             }
28
29             // Look to see if the proper child already exists
30             String segment = getUriSegment(uri, index);
31             if (segment == null) {
32                 throw new IllegalArgumentException("Invalid Uri (" + uri + ") used for observer");
33             }
34             int N = mChildren.size();
35             for (int i = 0; i < N; i++) {
36                 ObserverNode node = mChildren.get(i);
37                 if (node.mName.equals(segment)) {
38                     node.addObserverLocked(uri, index + 1, observer, notifyForDescendents, observersLock);
39                     return;
40                 }
41             }
42
43             // No child found, create one
44             ObserverNode node = new ObserverNode(segment);
45             mChildren.add(node);
46             node.addObserverLocked(uri, index + 1, observer, notifyForDescendents, observersLock);
47         }
48
49         ......
50     }
51
52     ......
53 }
```

ContentService在内部使用一个ObserverNode对象来保存一组监控相同数据的内容观察者，而每一个内容观察者都被封装成一个ObserverEntry对象保存在对应的ObserverNode对象的成员变量mObservers中。

前面提到，一个内容观察者所要监控的数据是使用一个URI来描述的。假设有两个内容观察者O1和O2，它们分别保存在ObserverNode对象N1和N2中，并且内容观察者 O2所要监控的数据的URI是以内容观察者 O1所要监控的数据的URI为前缀的，那么在ContentService中，ObserverNode对象N2就会保存在ObserverNode对象N1的成员变量mChildren中，表示ObserverNode对象N2是ObserverNode对象N1的一个子节点。ContentService就是通过这种形式将所有注册到它里面的内容观察者组织成一棵树的。

在ContentService中，每一个ObserverNode对象都有一个名称，保存在它的成员变量mName中。继续以上面的ObserverNode对象N1和N2为例，假设保存在它们里面的内容观察者所要监控的数据的URI分别为“content://my-authority”和“content://my-authority/item”，那么ObserverNode对象N1和N2的名称就分别为“my-authority”和“item”。

由于每一个ObserverNode对象的组织形式以及名称都与一个URI的结构相关，因此，ObserverNode类提供了两个成员函数countUriSegments和getUriSegment来辅助解析一个URI的结构，它们的实现如下所示。

frameworks/base/core/java/android/content/ContentService.java

```
01 public final class ContentService extends IContentService.Stub {
02     ......
03
04     public static final class ObserverNode {
05         ......
06
07         private String getUriSegment(Uri uri, int index) {
08             if (uri != null) {
09                 if (index == 0) {
10                     return uri.getAuthority();
11                 } else {
12                     return uri.getPathSegments().get(index - 1);
13                 }
14             } else {
15                 return null;
16             }
17         }
18
19         private int countUriSegments(Uri uri) {
20             if (uri == null) {
21                 return 0;
22             }
23             return uri.getPathSegments().size() + 1;
24         }
25
26         ......
27     }
28
29     ......
30 }
```

成员函数countUriSegments用来获得一个URI的路径长度，而成员函数getUriSegment用来获得一个URI的第index个路径的名称。例如，countUriSegments ("content://my-authority/item")的值为2，而getUriSegment("content://my-authority/item", 0)和getUriSegment("content://my-authority/item", 1)的值分别为“my-authority”和“item”。

理解了这些背景知识之后，接下来我们就结合参数observer所描述的一个内容观察者的注册过程来分析ObserverNode类的成员函数addObserverLocked的实现。

从前面的调用过程可以知道，内容观察者observer所要监控的数据的URI为“content://shy.luo.providers.articles/item”，在它的注册过程中，五个参数版本的成员函数addObserverLocked一共被调用了三次。

第一次是围绕根节点mRootNode来调用的，其中，参数uri和index的值分别为“content://shy.luo.providers.articles/item”和 0。这时候第24行的if语句为false，接下来第30行代码首先调用成员函数getUriSegment来获得“content://shy.luo.providers.articles/item”的第1个路径名称“shy.luo.providers.articles”，然后第35行到第41行的for循环在根节点mRootNode的成员变量mChildren中检查是否存在一个名称为“shy.luo.providers.articles”的子节点。如果存在，那么第38行代码就直接调用它的成员函数addObserverLocked来继续注册内容观察者observer；否则，第44行和第45行代码就需要先创建这个子节点，并且将它添加到根节点mRootNode的成员变量mChildren中，然后再调用它的成员函数addObserverLocked来继续注册内容观察者observer。

第二次是围绕名称为“shy.luo.providers.articles”的节点来调用的，其中，参数uri和index的值分别为“content://shy.luo.providers.articles/item”和 1。这时候第24行的if语句仍然为false，接下来第30行代码首先调用成员函数getUriSegment来获得“content://shy.luo.providers.articles/item”的第2个路径名称“item”，然后第35行到第41行的for循环在名称为“shy.luo.providers.articles”的节点的成员变量mChildren中检查是否存在一个名称为“item”的子节点。如果存在，那么第38行代码就直接调用它的成员函数addObserverLocked来继续注册内容观察者observer；否则，第44行和第45行代码就需要先创建这个子节点，并且将它添加到名称为“shy.luo.providers.articles”的节点的成员变量mChildren中，然后再调用它的成员函数addObserverLocked来继续注册内容观察者observer。

第三次是围绕名称为“item”的节点来调用的，其中，参数uri和index的值分别为“content://shy.luo.providers.articles/item”和 2。这时候第24行的if语句为 true，接下来第35行代码首先将内容观察者observer封装成一个ObserverEntry对象，然后将这个ObserverEntry对象添加到名称为“item”的节点的成员变量mObervers中，并且结束内容观察者observer的注册过程。这时候得到的内容观察者注册树的结构就如图10–22所示。

mROOTNode(''),which has an ObserverNode in mChildren.
----ObserverNode(''shy.luo.providers.articles''),which has an ObserverNode in mChildren.
----ObserverNode(''item'')which has an ObserverEntry in mObservers.

图10–22　内容观察者注册树的结构

至此，一个内容观察者的注册过程就分析完成了。接下来，我们继续分析Content Provider组件发送数据更新通知的过程。

10.4.2　发送数据更新通知

在前面的10.1.2小节中，当MainActivity组件向ArticlesProvider组件新增了一个博客文章条目之后，ArticlesProvider组件就会发送一个数据更新通知给所有注册了要监控它的数据更新的内容观察者，以便它们可以获得最新的博客文章信息。本节就结合这个使用场景，来分析Content Provider组件发送数据更新通知的过程。

Content Provider组件首先会将数据更新通知发送给ContentService，然后由ContentService将这个数据更新通知转发给目标内容观察者。这个过程一共分为8个步骤，下面我们就详细分析每一个步骤。

第1步到第4步如图10–23所示。

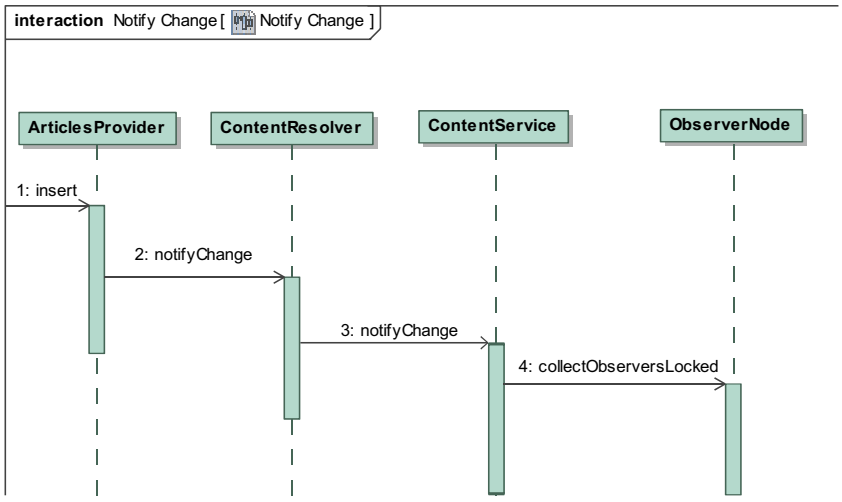


图10–23　数据更新通知的发送过程——Step 1至Step 4

Step 1：ArticlesProvider.insert

packages/experimental/ArticlesProvider/src/shy/luo/providers/articles/ArticlesProvider.java

```
01 public class ArticlesProvider extends ContentProvider {
02     ......
03
04     @Override
05     public Uri insert(Uri uri, ContentValues values) {
06         if(uriMatcher.match(uri) != Articles.ITEM) {
07             throw new IllegalArgumentException("Error Uri: " + uri);
08         }
09
10         SQLiteDatabase db = dbHelper.getWritableDatabase();
11
12         long id = db.insert(DB_TABLE, Articles.ID, values);
13         if(id < 0) {
14             throw new SQLiteException("Unable to insert " + values + " for " + uri);
15         }
16
17         Uri newUri = ContentUris.withAppendedId(uri, id);
18         resolver.notifyChange(newUri, null);
19
20         return newUri;
21     }
22
23     ......
24 }
```

参数uri所描述的URI的值必须等于“content://shy.luo.providers.articles/item”；否则，就不能往ArticlesProvider组件中新增博客文章条目了。

往ArticlesProvider组件中新增了一个博客文章条目之后，就会得到一个整数，用来描述新增的博客文章条目的ID。第17行代码将这个整数附加到参数uri所描述的一个URI的后面，从而获得一个形式为“content://shy.luo.providers.articles/item/#”的新URI。最后，第18行代码以这个新URI为参数来调用ArticlesProvider组件内部的一个ContentResolver对象resolver的成员函数notifyChange，以便可以向ContentService发送一个数据更新通知。

Step 2：ContentResolver.notifyChange

frameworks/base/core/java/android/content/ContentResolver.java

```
01 public abstract class ContentResolver {
02     ......
```

```
03
04      public void notifyChange(Uri uri, ContentObserver observer) {
05          notifyChange(uri, observer, true /* sync to network */);
06      }
07
08      public void notifyChange(Uri uri, ContentObserver observer, boolean syncToNetwork) {
09          try {
10              getContentService().notifyChange(
11                  uri, observer == null ? null : observer.getContentObserver(),
12                  observer != null && observer.deliverSelfNotifications(), syncToNetwork);
13          } catch (RemoteException e) {
14          }
15      }
16
17      ......
18  }
```

ContentResolver类有两个重载的成员函数notifyChange，其中一个有两个参数，另外一个有三个参数，前者调用后者向ContentService发送一个数据更新通知。

在三个参数版本的成员函数notifyChange中，第一个参数uri用来描述发生了数据更新的URI，第二个参数observer用来描述触发该数据更新的一个内容观察者，第三个参数syncToNetwork用来描述是否需要将该数据更新同步到网络中。

第10行到第12行代码首先调用成员函数getContentService来获得一个ContentService代理对象，然后调用这个代理对象的成员函数notifyChange向ContentService发送一个数据更新通知。

ContentService接收到这个数据更新通知之后，就会调用它的成员函数notifyChange来进一步处理。

Step 3：ContentService.notifyChange

frameworks/base/core/java/android/content/ContentService.java

```
01  public final class ContentService extends IContentService.Stub {
02      ......
03
04      public void notifyChange(Uri uri, IContentObserver observer,
05              boolean observerWantsSelfNotifications, boolean syncToNetwork) {
06          ......
07
08          try {
09              ArrayList<ObserverCall> calls = new ArrayList<ObserverCall>();
10              synchronized (mRootNode) {
11                  mRootNode.collectObserversLocked(uri, 0, observer, observerWantsSelfNotifications,
12                          calls);
13              }
14              final int numCalls = calls.size();
15              for (int i=0; i<numCalls; i++) {
16                  ObserverCall oc = calls.get(i);
17                  try {
18                      oc.mObserver.onChange(oc.mSelfNotify);
19                      ......
20                  } catch (RemoteException ex) {
21                      ......
22                  }
23              }
24              ......
25          } finally {
26              ......
27          }
28      }
29
30      ......
31  }
```

第11行和第12行代码首先调用成员变量mRootNode的成员函数collectObserverLocked来收集那些与参数uri对应的内容观察者，接着第15行到第23行的for循环再依次调用这些内容观察者的成员函数onChange向它们发送一个数据更新通知。

从前面10.4.1小节的内容可以知道，注册到ContentService中的内容观察者实际上是一个类型为Transport的Binder本地对象，因此，前面得到的内容观察者实际上是一系列类型为Transport的Binder代理对象。接下来，我们首先分析这些Transport代理对象的收集过程，然后分析它们所引用的Transport对象的成员函数onChange的实现。

Step 4：ObserverNode.collectObserversLocked

frameworks/base/core/java/android/content/ContentService.java

```
01 public final class ContentService extends IContentService.Stub {
02     ......
03
04     public static final class ObserverNode {
05         ......
06
07         private void collectMyObserversLocked(boolean leaf, IContentObserver observer,
08                 boolean selfNotify, ArrayList<ObserverCall> calls) {
09             int N = mObservers.size();
10             IBinder observerBinder = observer == null ? null : observer.asBinder();
11             for (int i = 0; i < N; i++) {
12                 ObserverEntry entry = mObservers.get(i);
13
14                 // Don't notify the observer if it sent the notification and isn't interesed
15                 // in self notifications
16                 if (entry.observer.asBinder() == observerBinder && !selfNotify) {
17                     continue;
18                 }
19
20                 // Make sure the observer is interested in the notification
21                 if (leaf || (!leaf && entry.notifyForDescendents)) {
22                     calls.add(new ObserverCall(this, entry.observer, selfNotify));
23                 }
24             }
25         }
26
27         public void collectObserversLocked(Uri uri, int index, IContentObserver observer,
28                 boolean selfNotify, ArrayList<ObserverCall> calls) {
29             String segment = null;
30             int segmentCount = countUriSegments(uri);
31             if (index >= segmentCount) {
32                 // This is the leaf node, notify all observers
33                 collectMyObserversLocked(true, observer, selfNotify, calls);
34             } else if (index < segmentCount){
35                 segment = getUriSegment(uri, index);
36                 // Notify any observers at this level who are interested in descendents
37                 collectMyObserversLocked(false, observer, selfNotify, calls);
38             }
39
40             int N = mChildren.size();
41             for (int i = 0; i < N; i++) {
42                 ObserverNode node = mChildren.get(i);
43                 if (segment == null || node.mName.equals(segment)) {
44                     // We found the child,
45                     node.collectObserversLocked(uri, index + 1, observer, selfNotify, calls);
46                     if (segment != null) {
47                         break;
48                     }
49                 }
50             }
51         }
52     }
53 }
```

假设目前只有MainActivity组件注册了一个内容观察者到ContentService中，下面我们就结合前面10.4.1小节中的图10-22来分析ObserverNode类的成员函数collectObserversLocked的实现。

从前面的调用过程可以知道，参数uri的值为“content://shy.luo.providers.articles/item/#”，在收集与它所对应的内容观察者的过程中，ObserverNode类的成员函数collectObserversLocked一共被调用了三次。

第一次是围绕根节点mRootNode来调用的，其中，参数uri和index的值分别为“content://shy.luo.providers.articles/item/#”和 0。这时候第34行的if语句为true，接下来第35行代码首先调用成员函数getUriSegment来获得“content://shy.luo.providers.articles/item/#”的第1个路径名称“shy.luo.providers.articles”，然后第37行代码调用成员函数collectMyObserversLocked来收集那些保存在根节点mRootNode中的目标内容观察者，最后第41行到第50行的for循环在根节点mRootNode中检查是否存在一个名称为“shy.luo.providers.articles”的子节点。如果存在，那么就会调用它的成员函数collectObserversLocked来继续收集目标内容观察者。

从前面的图10-22可以知道，根节点mRootNode的成员变量mObservers所描述的一个内容观察者列表为空，因此，这时候调用它的成员函数collectMyObserversLocked就不会收集到目标内容观察者。

第二次是围绕名称为“shy.luo.providers.articles”的节点来调用的，其中，参数uri和index的值分别为“content://shy.luo.providers.articles/item/#”和 1。这时候第34行的if语句仍然为true，接下来第35行代码首先调用成员函数getUriSegment来获得“content://shy.luo.providers.articles/item/#”的第2个路径名称“item”，然后第37行代码调用成员函数collectMyObserversLocked来收集那些保存在名称为“shy.luo.providers.articles”的节点中的目标内容观察者，最后第41行到第50行的for循环在名称为“shy.luo.providers.articles”的节点中检查是否存在一个名称为“item”的子节点。如果存在，那么就会调用它的成员函数collectObserversLocked来继续收集目标内容观察者。

从前面的图10-22可以知道，名称为“shy.luo.providers.articles”的节点的成员变量mObservers所描述的一个内容观察者列表也为空，因此，这时候调用它的成员函数collectMyObserversLocked也不会收集到目标内容观察者。

第三次是围绕名称为“item”的节点来调用的，其中，参数uri和index的值分别为“content://shy.luo.providers.articles/item/#”和 2。这时候第34行的if语句仍然为true，接下来第35行代码首先调用成员函数getUriSegment来获得“content://shy.luo.providers.articles/item/#”的第3个路径名称“#”，然后第37行代码调用成员函数collectMyObserversLocked来收集那些保存在名称为“item”的节点中的目标内容观察者，最后第41行到第50行的for循环在名称为“item”的节点中检查是否存在一个名称为“#”的子节点。如果存在，那么就会调用它的成员函数collectObserversLocked来继续收集目标内容观察者。由于名称为“item”的节点并不存在一个名称为“#”的子节点，因此，接下来就不会再调用ObserverNode类的成员函数collectObserversLocked来收集目标内容观察者了。

从前面的图10-22可以知道，名称为“item”的节点的成员变量mObservers中保存了一个内容观察者，这个内容观察者是在前面的10.4.1小节中注册的。接下来，我们就分析成员函数collectMyObserversLocked的实现，看看这时候调用成员函数collectMyObserversLocked是否能收集到目标内容观察者。

在成员函数collectMyObserversLocked中，第一个参数leaf用来描述当前是否正在收集一个URI的叶子路径所对应的内容观察者；第二个参数observer用来描述触发数据更新的内容观察者；第三个参数selfNotify用来描述是否要向触发数据更新的内容观察者发送一个数据更新通知。第11行到第24行的for循环遍历保存在当前节点的成员变量mObservers中的每一个内容观察者：

（1）如果这个内容观察者就是触发数据更新的内容观察者，并且参数selfNotify的值等于false，即第16行的if语句为true，那么就不会将它作为一个目标内容观察者。

（2）如果当前正在收集的是一个URI的叶子路径所对应的内容观察者，即第21行的if语句的第一

个表达式为true，那么这个内容观察者就是一个目标内容观察者。因此，第22行代码就会将它封装成一个ObserverCall对象添加到目标内容观察者列表calls中。

（3）如果当前正在收集的不是一个URI的叶子路径所对应的内容观察者，但是这个内容观察者除了需要监控它所指定的URI的数据更新之外，还需要监控以它所指定的URI为前缀的其他URI的数据更新，即第21行的if语句的第二个表达式为true，那么这个内容观察者也是一个目标内容观察者。因此，第22行代码也会将它封装成一个ObserverCall对象添加到目标内容观察者列表calls中。

虽然“item”不是“content://shy.luo.providers.articles/item/#”的叶子路径，但是从前面10.4.1小节的内容可以知道，保存在名称为“item”的节点中的内容观察者是需要监控以它所指定的URI为前缀的其他URI的数据更新的，因此，这个内容观察者会被添加到目标内容观察者列表calls中。

这一步执行完成之后，返回到前面的Step 3中，即ContentService类的成员函数notifyChange中，这时候ContentService就收集到所有的目标内容观察者了。在我们这个场景中，ContentService收集到的目标内容观察者只有一个，即为MainActivity组件所注册的内容观察者，接下来第5步到第8步就会向它发送一个数据更新通知，如图10–24所示。

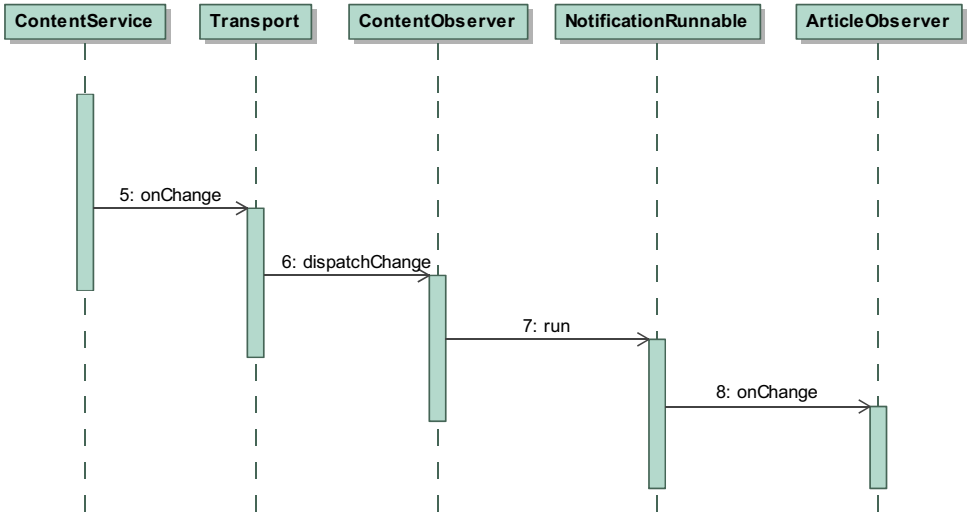


图10–24　数据更新通知的发送过程——Step 5至Step 8

Step 5：Transport.onChange

frameworks/base/core/java/android/database/ContentObserver.java

```
01 public abstract class ContentObserver {
02     ......
03
04     private static final class Transport extends IContentObserver.Stub {
05         ContentObserver mContentObserver;
06         ......
07
08         public void onChange(boolean selfChange) {
09             ContentObserver contentObserver = mContentObserver;
10             if (contentObserver != null) {
11                 contentObserver.dispatchChange(selfChange);
12             }
13         }
14
15         ......
16     }
17
18     ......
19 }
```

一个Transport对象接收到ContentService发送过来的一个数据更新通知之后，会将这个数据更新通知转发给与它关联的一个内容观察者来处理，即调用它的成员变量mContentObserver的成员函数dispatchChange来处理。

Step 6：ContentObserver.dispatchChange

frameworks/base/core/java/android/database/ContentObserver.java

```
01 public abstract class ContentObserver {
02     ......
03
04     /* package */ Handler mHandler;
05     ......
06
07     public ContentObserver(Handler handler) {
08         mHandler = handler;
09     }
10     ......
11
12     public final void dispatchChange(boolean selfChange) {
13         if (mHandler == null) {
14             onChange(selfChange);
15         } else {
16             mHandler.post(new NotificationRunnable(selfChange));
17         }
18     }
19 }
```

ContentObserver类有一个类型为Handler的成员变量mHandler，它是用来向一个线程的消息队列发送消息的。当ContentObserver类接收到ContentService发送过来的一个数据更新通知时，如果它的成员变量mHandler的值为null，那么接收线程就会直接调用它的成员函数onChange来处理该数据更新；否则，接收线程就会先将这个数据更新封装成一个类型为NotificationRunnable的消息，然后将这个消息发送到一个指定的线程的消息队列中去处理。这个指定的线程一般就是应用程序进程的主线程了，这样做的好处一方面是可以让接收线程尽快返回到Binder线程池中去处理其他进程间通信请求；另一方面是使得ContentObserver类在处理数据更新时可以执行界面相关的操作。

从前面10.4.1小节的Step 1可以知道，MainActivity组件所注册的内容观察者指定了一个Handler对象，并且这个Handler对象是用来向MainActivity组件所运行在的应用程序进程的主线程的消息队列发送消息的。因此，接下来第16行代码就会将接收到的一个数据更新通知封装成一个类型为NotificationRunnable的消息，然后发送到MainActivity组件所运行在的应用程序进程的主线程的消息队列中。这个消息最终是在NotificationRunnable类的成员函数run中处理的。

Step 7：NotificationRunnable.run

frameworks/base/core/java/android/database/ContentObserver.java

```
01 public abstract class ContentObserver {
02     ......
03
04     private final class NotificationRunnable implements Runnable {
05         private boolean mSelf;
06
07         public NotificationRunnable(boolean self) {
08             mSelf = self;
09         }
10
11         public void run() {
```

```
12              ContentObserver.this.onChange(mSelf);
13          }
14      }
15
16      ......
17  }
```

NotificationRunnable类的成员函数run直接调用了其外部类ContentObserver的成员函数onChange来接收它所封装的一个数据更新通知。我们在实现自己的内容观察者时，需要重写父类ContentObserver的成员函数onChange，因此，第12行代码实际上是调用ContentObserver子类的成员函数onChange来接收一个数据更新通知。在我们这个场景中，这个ContentObserver子类就是在MainActivity组件内部实现的ArticlesObserver类，因此，接下来就会调用ArticlesObserver类的成员函数onChange来接收一个数据更新通知。

Step 8：ArticleObserver.onChange

packages/experimental/Article/src/shy/luo/article/MainActivity.java

```
01 public class MainActivity extends Activity implements View.OnClickListener, AdapterView.
OnItemClickListener {
02      ......
03
04      private ArticleAdapter adapter = null;
05      ......
06
07      private class ArticleObserver extends ContentObserver {
08          ......
09
10          @Override
11          public void onChange(boolean selfChange) {
12              adapter.notifyDataSetChanged();
13          }
14      }
15
16      ......
17  }
```

当ArticleObserver类的成员函数onChange被调用时，MainActivity组件就知道ArticlesProvider组件中的博客文章信息发生更新了，这时候它就会通过内部的一个ArticleAdapter 对象adapter来更新用户界面上的博客文章列表，以便可以将ArticlesProvider组件中的最新博客文章信息显示出来。

至此，Content Provider组件发送数据更新通知的过程就分析完成了。结合前面10.4.1小节所介绍的内容观察者的注册过程，我们就可以对Content Provider组件的数据更新通知机制有一个清晰的认识。

第 11 章

Zygote和System进程的启动过程

在Android系统中，所有的应用程序进程，以及用来运行系统关键服务的System进程都是由Zygote进程负责创建的，因此，我们将它称为进程孵化器。Zygote进程是通过复制自身的方式来创建System进程和应用程序进程的。由于Zygote进程在启动时会在内部创建一个虚拟机实例，因此，通过复制Zygote进程而得到的System进程和应用程序进程可以快速地在内部获得一个虚拟机实例拷贝。

Zygote进程在启动完成之后，会马上将System进程启动起来，以便它可以将系统的关键服务启动起来。这些关键服务包括我们在前面第7章到第10章中经常碰到的Activity管理服务ActivityManagerService和Content管理服务ContentService，以及在后面的第14章和第16章中要用到的Window管理服务WindowManagerService和Package管理服务PackageManagerService。

下面我们首先介绍Zygote进程的启动脚本，然后分析它和System进程的启动过程。

11.1 Zygote进程的启动脚本

Zygote进程是由Android系统的第一个进程init启动起来的。init进程是在内核加载完成之后就启动起来的，它在启动的过程中，会读取根目录下的一个脚本文件init.rc，以便可以将其他需要开机启动的进程也一起启动起来。

Zygote进程在脚本文件init.rc中的启动脚本如下所示。

system/core/rootdir/init.rc

```
1 service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
2     socket zygote stream 666
```

第1行表示Zygote进程是以服务的形式启动的，并且它所对应的应用程序文件为/system/bin/app_process。接下来的四个选项是Zygote进程的启动参数，其中，最后一个参数“--start-system-server”表示Zygote进程在启动完成之后，需要马上将System进程也启动起来。

第2行表示Zygote进程在启动的过程中，需要在内部创建一个名称为“zygote”的Socket。这个Socket是用来执行进程间通信的，它的访问权限被设置为666，即所有用户都可以对它进行读和写。在后面的第12章中，我们就会看到，运行在System进程中的Activity管理服务ActivityManagerService就是通过这个Socket来请求Zygote进程创建新的应用程序进程的。

接下来，我们就主要分析Zygote进程是如何在启动的过程中创建一个名称为“zygote”的Socket的。

由于Zygote进程在脚本文件init.rc中配置了以服务的形式来启动，因此，init进程在启动时，就会调用函数service_start来启动它，如下所示。

system/core/init/init.c

```
01 void service_start(struct service *svc, const char *dynamic_args)
02 {
03     ......
04
05     pid_t pid;
06     ......
07
08     pid = fork();
09
10     if (pid == 0) {
11         struct socketinfo *si;
12         ......
13
14         for (si = svc->sockets; si; si = si->next) {
15             int socket_type = (
16                 !strcmp(si->type, "stream") ? SOCK_STREAM :
17                 (!strcmp(si->type, "dgram") ? SOCK_DGRAM : SOCK_SEQPACKET));
18             int s = create_socket(si->name, socket_type,
19                 si->perm, si->uid, si->gid);
20             if (s >= 0) {
21                 publish_socket(si->name, s);
22             }
23         }
24         ......
25
26         if (!dynamic_args) {
27             if (execve(svc->args[0], (char**) svc->args, (char**) ENV) < 0) {
28                 ERROR("cannot execve('%s'): %s\n", svc->args[0], strerror(errno));
29             }
30         } else {
31             char *arg_ptrs[INIT_PARSER_MAXARGS+1];
32             int arg_idx = svc->nargs;
33             char *tmp = strdup(dynamic_args);
34             char *next = tmp;
35             char *bword;
36
37             /* Copy the static arguments */
38             memcpy(arg_ptrs, svc->args, (svc->nargs * sizeof(char *)));
39
40             while((bword = strsep(&next, " "))) {
41                 arg_ptrs[arg_idx++] = bword;
42                 if (arg_idx == INIT_PARSER_MAXARGS)
43                     break;
44             }
45             arg_ptrs[arg_idx] = '\0';
46             execve(svc->args[0], (char**) arg_ptrs, (char**) ENV);
47         }
48         _exit(127);
49     }
50
51     ......
52 }
```

参数svc用来描述一个即将要启动的服务；参数dynamic_args用来描述这个服务的动态启动参数列表。动态启动参数是相对于在脚本文件init.rc中配置的静态启动参数而言的，它的值可以为NULL。参数svc指向了一个service结构体，它的成员变量sockets和args分别保存了即将要启动的服务的Socket列表和静态启动参数列表。

第8行代码首先调用函数fork来创建一个子进程。函数fork调用成功之后，会有两次返回，其中一次是在当前进程中返回，而另外一次是在新创建的子进程中返回。如果函数fork的返回值等于0，那

么就说明它是在新创建的子进程中返回的。在我们这个场景中，新创建的子进程即为Zygote进程，我们主要分析它从函数fork返回来之后的执行过程。

第14行到第23行的for循环依次遍历在脚本文件init.rc中为Zygote进程配置的Socket列表。列表中的每一个Socket都是通过函数create_socket创建的。函数create_socket成功地创建了一个Socket之后，就会返回一个文件描述符。这个文件描述符最后会通过函数publish_socket发布到系统中。

将要在Zygote进程中加载的应用程序文件的名称保存在参数svc->args[0]中，它的值等于“/system/bin/app_process”。第26行的if语句检查Zygote进程是否有动态设置的启动参数，即参数dynamic_args的值是否等于NULL。如果没有，那么第27行代码就直接在Zygote进程中加载应用程序文件/system/bin/app_process，并且将Zygote进程的静态启动参数列表传递给它；否则，第31行到第46行代码首先将Zygote进程的静态启动参数和动态启动参数合并到数组arg_ptrs中，然后再在Zygote进程中加载应用程序文件/system/bin/app_process，并且将数组arg_ptrs传递给它。

接下来，我们主要分析函数create_socket和publish_socket的实现，以便可以了解它们是如何为Zygote进程创建一个名称为“zygote”的Socket的。

函数create_socket的实现如下所示。

system/core/init/util.c

```
01 /*
02  * create_socket - creates a Unix domain socket in ANDROID_SOCKET_DIR
03  * ("/dev/socket") as dictated in init.rc. This socket is inherited by the
04  * daemon. We communicate the file descriptor's value via the environment
05  * variable ANDROID_SOCKET_ENV_PREFIX<name> ("ANDROID_SOCKET_foo").
06  */
07 int create_socket(const char *name, int type, mode_t perm, uid_t uid, gid_t gid)
08 {
09     struct sockaddr_un addr;
10     int fd, ret;
11
12     fd = socket(PF_UNIX, type, 0);
13     ......
14
15     memset(&addr, 0 , sizeof(addr));
16     addr.sun_family = AF_UNIX;
17     snprintf(addr.sun_path, sizeof(addr.sun_path), ANDROID_SOCKET_DIR"/%s",
18              name);
19     ......
20
21     ret = bind(fd, (struct sockaddr *) &addr, sizeof (addr));
22     ......
23
24     chown(addr.sun_path, uid, gid);
25     chmod(addr.sun_path, perm);
26     ......
27
28     return fd;
29 }
```

第12行代码首先调用函数socket来创建一个Socket，这个Socket使用文件描述符fd来描述。接着第15行到第18行代码为这个Socket创建了一个类型为AF_UNIX 的Socket地址addr，这个Socket地址有一个对应的设备文件，即ANDROID_SOCKET_DIR/name[1]。

1　AF_UNIX类型的Socket是用来执行本地进程间通信的，具体可以参考*Linux内核源代码情景分析一*书的第7章——基于Socket的进程间通信。

从Zygote进程的启动脚本可以知道，参数name的值等于“zygote”；而ANDROID_SOCKET_DIR是一个宏，它的值等于“/dev/socket”。因此，第21行代码调用函数bind将文件描述符fd所描述的Socket与Socket地址addr绑定起来之后，我们就可以得到一个设备文件/dev/socket/zygote。

最后，第24行和第25行代码分别调用函数chown和chmod来设置设备文件/dev/socket/zygote的用户ID、用户组ID以及访问权限。从Zygote进程的启动脚本可以知道，参数perm的值为666，因此，设备文件/dev/socket/zygote是可以被任意用户访问的。在后面的第12章中，我们就会看到，Activity管理服务ActivityManagerService在请求Zygote进程创建新的应用程序进程之前，首先会通过打开这个设备文件来创建一个连接到Zygote进程的Client端Socket。

函数publish_socket的实现如下所示。

system/core/init/init.c

```
01 static void publish_socket(const char *name, int fd)
02 {
03     char key[64] = ANDROID_SOCKET_ENV_PREFIX;
04     char val[64];
05
06     strlcpy(key + sizeof(ANDROID_SOCKET_ENV_PREFIX) - 1,
07             name,
08             sizeof(key) - sizeof(ANDROID_SOCKET_ENV_PREFIX));
09     snprintf(val, sizeof(val), "%d", fd);
10     add_environment(key, val);
11
12     /* make sure we don't close-on-exec */
13     fcntl(fd, F_SETFD, 0);
14 }
```

第6行到第8行代码将宏ANDROID_SOCKET_ENV_PREFIX和参数name所描述的两个字符串连接在一起，保存在字符串key中。宏ANDROID_SOCKET_ENV_PREFIX的值为“ANDROID_SOCKET_”，而参数name的值为“zygote”，因此，前面得到的字符串key的值就为“ANDROID_SOCKET_zygote”。

参数fd是一个文件描述符，指向了前面所创建的一个Socket。第9行代码首先将参数fd格式化为一个字符串，并且保存在变量val中。接着第10行代码在系统中增加一个名称为key的环境变量，并且将它的值设置为val。这就相当于将前面所创建的一个Socket的文件描述符保存在名称为“ANDROID_SOCKET_zygote”的环境变量中。在接下来的11.2小节中，我们就会看到，Zygote进程在启动的过程中，会根据这个环境变量的值来创建一个Server端Socket，等到启动完成之后，再在这个Server端Socket上等待Activity管理服务ActivityManagerService请求它创建新的应用程序进程。

至此，我们就分析完成Zygote进程的启动脚本了。下面我们继续分析Zygote进程的启动过程。

11.2 Zygote进程的启动过程

从前面11.1小节的内容可以知道，在Zygote进程中加载的应用程序文件为system/bin/app_process。因此，接下来我们就从这个应用程序文件的入口函数main开始分析Zygote进程的启动过程，如图11-1所示。

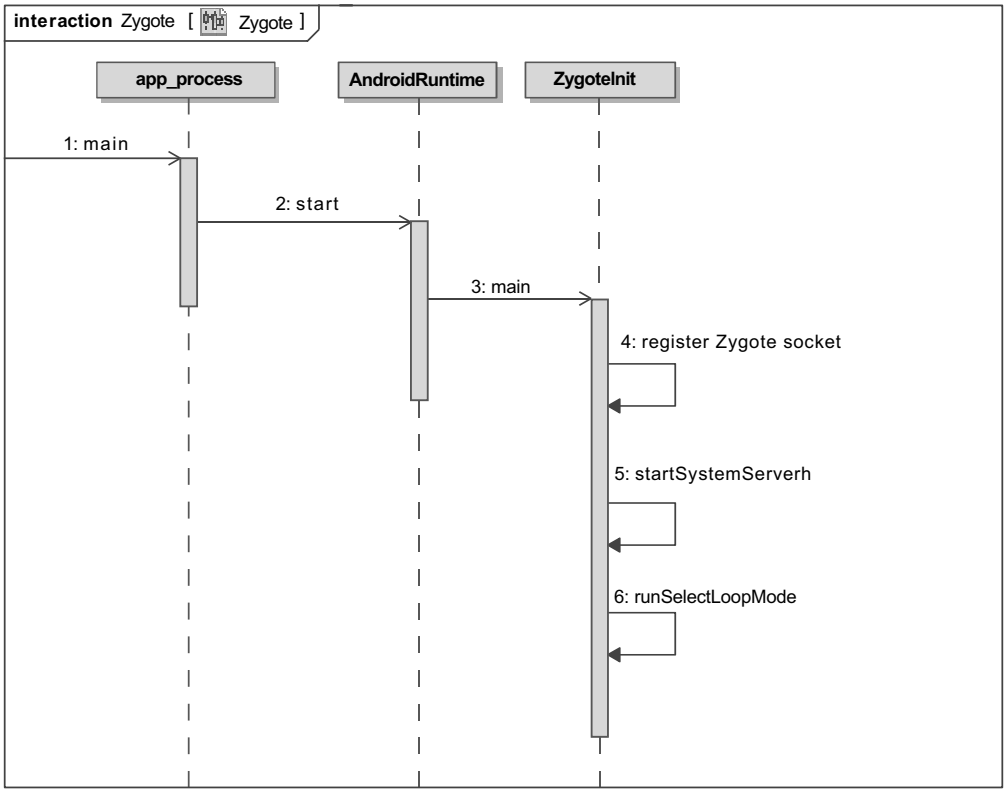


图11-1　Zygote进程的启动过程

这个过程一共分为6个步骤，下面我们就详细分析每一个步骤。

Step 1：app_process.main

frameworks/base/cmds/app_process/app_main.cpp

```
01 int main(int argc, const char* const argv[])
02 {
03     // These are global variables in ProcessState.cpp
04     mArgC = argc;
05     mArgV = argv;
06
07     mArgLen = 0;
08     for (int i=0; i<argc; i++) {
09         mArgLen += strlen(argv[i]) + 1;
10     }
11     mArgLen--;
12
13     AppRuntime runtime;
14     const char *arg;
15     argv0 = argv[0];
16
17     // Process command line arguments
18     // ignore argv[0]
19     argc--;
20     argv++;
21
22     // Everything up to '--' or first non '-' arg goes to the vm
23
24     int i = runtime.addVmArguments(argc, argv);
25
26     // Next arg is parent directory
```

```
27      if (i < argc) {
28          runtime.mParentDir = argv[i++];
29      }
30
31      // Next arg is startup classname or "--zygote"
32      if (i < argc) {
33          arg = argv[i++];
34          if (0 == strcmp("--zygote", arg)) {
35              bool startSystemServer = (i < argc) ?
36                  strcmp(argv[i], "--start-system-server") == 0 : false;
37              setArgv0(argv0, "zygote");
38              set_process_name("zygote");
39              runtime.start("com.android.internal.os.ZygoteInit",
40                  startSystemServer);
41          }
42          ......
43      }
44      ......
45  }
```

第13行代码创建了一个AppRuntime对象runtime，接下来Zygote进程就是通过它来进一步启动的，我们重点分析第34行到第41行代码。

第34行的if语句检查应用程序app_process的启动参数arg是否包含了一个“--zygote”选项。如果包含了，那么就说明这时候应用程序app_process是在Zygote进程中启动的。

从前面11.1小节的内容可以知道，Zygote进程在加载应用程序app_process时，传递过来的启动参数arg是包含了一个“--zygote”选项的。因此，第38行代码首先会调用函数set_process_name将当前进程的名称设置为“zygote”，然后第39行和第40行代码调用AppRuntime对象runtime的成员函数start来进一步启动Zygote进程。

从前面11.1小节的内容还可以知道，Zygote进程传递给应用程序app_process的启动参数arg还包含了一个“--start-system-server”选项。因此，在调用AppRuntime对象runtime的成员函数start时，传进去的第二个参数startSystemServer的值等于true，表示Zygote进程在启动完成之后，需要将System进程启动起来。

AppRuntime类的成员函数start是从其父类AndroidRuntime继承下来的，因此，接下来我们就继续分析AndroidRuntime类的成员函数start的实现。

Step 2：AndroidRuntime.start

frameworks/base/core/jni/AndroidRuntime.cpp

```
01  void AndroidRuntime::start(const char* className, const bool startSystemServer)
02  {
03      ......
04
05      char* slashClassName = NULL;
06      char* cp;
07      JNIEnv* env;
08      ......
09
10      /* start the virtual machine */
11      if (startVm(&mJavaVM, &env) != 0)
12          goto bail;
13
14      /*
15      * Register android functions.
16      */
17      if (startReg(env) < 0) {
18          LOGE("Unable to register all android natives\n");
```

```
19          goto bail;
20      }
21
22      /*
23      * We want to call main() with a String array with arguments in it.
24      * At present we only have one argument, the class name.  Create an
25      * array to hold it.
26      */
27      jclass stringClass;
28      jobjectArray strArray;
29      jstring classNameStr;
30      jstring startSystemServerStr;
31      stringClass = env->FindClass("java/lang/String");
32      assert(stringClass != NULL);
33      strArray = env->NewObjectArray(2, stringClass, NULL);
34      assert(strArray != NULL);
35      classNameStr = env->NewStringUTF(className);
36      assert(classNameStr != NULL);
37      env->SetObjectArrayElement(strArray, 0, classNameStr);
38      startSystemServerStr = env->NewStringUTF(startSystemServer ?
39          "true" : "false");
40      env->SetObjectArrayElement(strArray, 1, startSystemServerStr);
41
42      /*
43      * Start VM.  This thread becomes the main thread of the VM, and will
44      * not return until the VM exits.
45      */
46      jclass startClass;
47      jmethodID startMeth;
48
49      slashClassName = strdup(className);
50      for (cp = slashClassName; *cp != '\0'; cp++)
51          if (*cp == '.')
52              *cp = '/';
53
54      startClass = env->FindClass(slashClassName);
55      if (startClass == NULL) {
56          ......
57      } else {
58          startMeth = env->GetStaticMethodID(startClass, "main",
59              "([Ljava/lang/String;)V");
60          if (startMeth == NULL) {
61              ......
62          } else {
63              env->CallStaticVoidMethod(startClass, startMeth, strArray);
64              ......
65          }
66      }
67
68      ......
69  }
```

从前面的调用过程可以知道，参数className的值等于“com.android.internal.os.ZygoteInit”；参数startSystemServer的值等于true。

第11行代码首先调用成员函数startVM在Zygote进程中创建一个虚拟机实例，然后第17行代码调用成员函数startReg在这个虚拟机实例中注册一系列JNI方法，最后第63行代码调用com.android.internal.os.ZygoteInit类的静态成员函数main来进一步启动Zygote进程。

注意

在调用com.android.internal.os.ZygoteInit类的静态成员函数main时，传进去的参数strArray是一个字符串数组，它的大小为2，其中，第二个元素的值等于true。

接下来，我们就继续分析com.android.internal.os.ZygoteInit类的静态成员函数main的实现。

Step 3：ZygoteInit.main

frameworks/base/core/java/com/android/internal/os/ZygoteInit.java

```
01 public class ZygoteInit {
02     ......
03
04     private static final boolean ZYGOTE_FORK_MODE = false;
05     ......
06
07     public static void main(String argv[]) {
08         try {
09             ......
10
11             registerZygoteSocket();
12             ......
13
14             if (argv[1].equals("true")) {
15                 startSystemServer();
16             }
17             ......
18
19             if (ZYGOTE_FORK_MODE) {
20                 runForkMode();
21             } else {
22                 runSelectLoopMode();
23             }
24
25             ......
26         } catch (MethodAndArgsCaller caller) {
27             ......
28         } catch (RuntimeException ex) {
29             ......
30         }
31     }
32
33     ......
34 }
```

第11行代码首先调用ZygoteInit类的静态成员函数registerZygoteSocket来创建一个Server端Socket，这个Server端Socket是用来等待Activity管理服务ActivityManagerService请求Zygote进程创建新的应用程序进程的。

从前面的调用过程可以知道，参数argv[1]的值等于true，因此，第15行代码就会调用ZygoteInit类的静态成员函数startSystemServer来启动System进程，以便它可以将系统的关键服务启动起来。

ZYGOTE_FORK_MODE是ZygoteInit类的一个静态成员变量，它的值被设置为false，因此，第22行代码就会调用ZygoteInit类的静态成员函数runSelectLoopMode来等待Activity管理服务ActivityManagerService请求Zygote进程创建新的应用程序进程。

如果ZygoteInit类的静态成员变量ZYGOTE_FORK_MODE的值为true，那么Zygote进程就会调用ZygoteInit类的静态成员函数runForkMode来等待Activity管理服务ActivityManagerService请求Zygote进程创建新的应用程序进程。不过，在这种情况下，对于每一个请求，Zygote进程都会创建一个新的进程来处理，这样会浪费大量的进程资源。因此，为了让所有的创建应用程序进程请求都在同一个进程中处理，我们将ZygoteInit类的静态成员变量ZYGOTE_FORK_MODE的值设置为false。

接下来，我们就分别分析ZygoteInit类的静态成员函数registerZygoteSocket、startSystemServer和runSelectLoopMode的实现。

Step 4：ZygoteInit.registerZygoteSocket

frameworks/base/core/java/com/android/internal/os/ZygoteInit.java

```
01 public class ZygoteInit {
02     ......
03
04     private static LocalServerSocket sServerSocket;
05     ......
06
07     private static final String ANDROID_SOCKET_ENV = "ANDROID_SOCKET_zygote";
08     ......
09
10     private static void registerZygoteSocket() {
11         if (sServerSocket == null) {
12             int fileDesc;
13             try {
14                 String env = System.getenv(ANDROID_SOCKET_ENV);
15                 fileDesc = Integer.parseInt(env);
16             } catch (RuntimeException ex) {
17                 ......
18             }
19
20             try {
21                 sServerSocket = new LocalServerSocket(
22                     createFileDescriptor(fileDesc));
23             } catch (IOException ex) {
24                 .......
25             }
26         }
27     }
28
29     ......
30 }
```

第14行代码首先获得一个名称为“ANDROID_SOCKET_zygote”的环境变量的值，接着第15行代码将它转换为一个文件描述符。从前面11.1小节的内容可以知道，这个文件描述符是用来描述一个类型为AF_UNIX的Socket的，接下来第21行和第22行代码根据这个文件描述符来创建一个Server端Socket，并且保存在ZygoteInit的静态成员变量sServerSocket中。Zygote进程将System进程启动起来之后，就会在这个Server端Socket上等待Activity管理服务ActivityManagerService请求Zygote进程创建新的应用程序进程。

这一步执行完成之后，回到前面的Step 3中，即ZygoteInit类的静态成员函数main中，接下来就会调用ZygoteInit类的静态成员函数startSystemServer来启动System进程。

Step 5：ZygoteInit.startSystemServer

frameworks/base/core/java/com/android/internal/os/ZygoteInit.java

```
01 public class ZygoteInit {
02     ......
03
04     private static boolean startSystemServer()
05             throws MethodAndArgsCaller, RuntimeException {
06         /* Hardcoded command line to start the system server */
07         String args[] = {
08             "--setuid=1000",
09             "--setgid=1000",
10             "--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,3001,3002,3003",
11             "--capabilities=130104352,130104352",
12             "--runtime-init",
13             "--nice-name=system_server",
```

```
14              "com.android.server.SystemServer",
15          };
16          ZygoteConnection.Arguments parsedArgs = null;
17
18          int pid;
19
20          try {
21              parsedArgs = new ZygoteConnection.Arguments(args);
22              ......
23
24              /* Request to fork the system server process */
25              pid = Zygote.forkSystemServer(
26                  parsedArgs.uid, parsedArgs.gid,
27                  parsedArgs.gids, debugFlags, null,
28                  parsedArgs.permittedCapabilities,
29                  parsedArgs.effectiveCapabilities);
30          } catch (IllegalArgumentException ex) {
31              ......
32          }
33
34          /* For child process */
35          if (pid == 0) {
36              handleSystemServerProcess(parsedArgs);
37          }
38
39          return true;
40      }
41
42      ......
43  }
```

第7行到第15行代码首先创建一个字符串数组args，用来保存System进程的启动参数。接着第25行到第29行代码就会调用Zygote类的静态成员函数forkSystemServer来创建一个子进程，这个子进程就是Android系统的System进程。从这里就可以看出，Android系统的System进程的用户ID和用户组ID均被设置为1000，并且它还具有用户组1001~1010、1018以及3001~3003的权限。

Zygote类的静态成员函数forkSystemServer最终是通过函数fork在当前进程中创建一个子进程的，因此，当它的返回值等于0时，就表示是在新创建的子进程中执行的，这时候ZygoteInit类就会调用静态成员函数handleSystemServerProcess来启动System进程。在接下来的11.3小节中，我们再分析System进程的启动过程。

这一步执行完成之后，返回到前面的Step 3中，即ZygoteInit类的静态成员函数main中，接下来就会调用ZygoteInit类的静态成员函数runSelectLoopMode来等待Activity管理服务ActivityManagerService请求Zygote进程创建新的应用程序进程。

Step 6：ZygoteInit.runSelectLoopMode

frameworks/base/core/java/com/android/internal/os/ZygoteInit.java

```
01 public class ZygoteInit {
02      ......
03
04      private static void runSelectLoopMode() throws MethodAndArgsCaller {
05          ArrayList<FileDescriptor> fds = new ArrayList();
06          ArrayList<ZygoteConnection> peers = new ArrayList();
07          FileDescriptor[] fdArray = new FileDescriptor[4];
08
09          fds.add(sServerSocket.getFileDescriptor());
10          peers.add(null);
11          ......
12
```

```
13          while (true) {
14              int index;
15              ......
16
17              try {
18                  fdArray = fds.toArray(fdArray);
19                  index = selectReadable(fdArray);
20              } catch (IOException ex) {
21                  throw new RuntimeException("Error in select()", ex);
22              }
23
24              if (index < 0) {
25                  throw new RuntimeException("Error in select()");
26              } else if (index == 0) {
27                  ZygoteConnection newPeer = acceptCommandPeer();
28                  peers.add(newPeer);
29                  fds.add(newPeer.getFileDesciptor());
30              } else {
31                  boolean done;
32                  done = peers.get(index).runOnce();
33
34                  if (done) {
35                      peers.remove(index);
36                      fds.remove(index);
37                  }
38              }
39          }
40      }
41
42      ......
43  }
```

第7行代码首先创建了一个大小为4的Socket文件描述符数组fdArray，表示Zygote进程最多能同时处理4个Socket连接。接着第9行代码将ZygoteInit 类的静态成员变量sServerSocket所描述的一个Socket的文件描述符添加到Socket文件描述符列表fds中。

第13行到第39行的无限循环用来等待Activity管理服务ActivityManagerService请求Zygote进程创建新的应用程序进程。第18行代码首先将保存在Socket文件描述符列表fds中的Socket文件描述符转移到Socket文件描述符数组fdArray中。接着第19行代码调用静态成员函数selectReadable来监控保存在这个数组中的Socket是否有数据可读。一个Socket有数据可读就意味着它接收到了一个连接或者一个请求。

如果变量index的值等于0，即第26行的if语句为true，那么就说明Activity管理服务ActivityManagerService通过ZygoteInit 类的静态成员变量sServerSocket所描述的一个Socket与Zygote进程建立了新的连接。

注意

这时候Activity管理服务ActivityManagerService只是与Zygote进程建立了一个连接，但是还没有请求Zygote进程为它创建一个新的应用程序进程。因此，第27行到第29行代码只是将这个连接添加到Socket连接列表peers中，并且将用来描述这个连接的一个Socket文件描述符添加到Socket文件描述符列表fds中，以便接下来可以接收到Activity管理服务ActivityManagerService发送过来的创建应用程序进程请求。

如果变量index的值大于0，即第24行和第26行的if语句均为false，那么就说明Activity管理服务ActivityManagerService向Zygote进程发送了一个创建应用程序进程的请求。因此，第32行代码就会调用前面已经建立起来的一个连接（ZygoteConnection）的成员函数runOnce来创建一个新的应用程序进程，并且第35行和第36行代码分别将这个连接从Socket连接列表peers和Socket文件描述符列表

fds中删除，因为这时候Zygote进程已经处理完成这个连接了。在后面的第12章中，我们再详细分析ZygoteConnection类的成员函数runOnce的实现。

至此，Zygote进程的启动过程就分析完成了。接下来我们继续分析System进程的启动过程。

11.3 System进程的启动过程

从前面11.2小节的内容可以知道，System进程是在ZygoteInit类的静态成员函数handleSystemServerProcess中开始启动的，它的启动过程如图11-2所示。

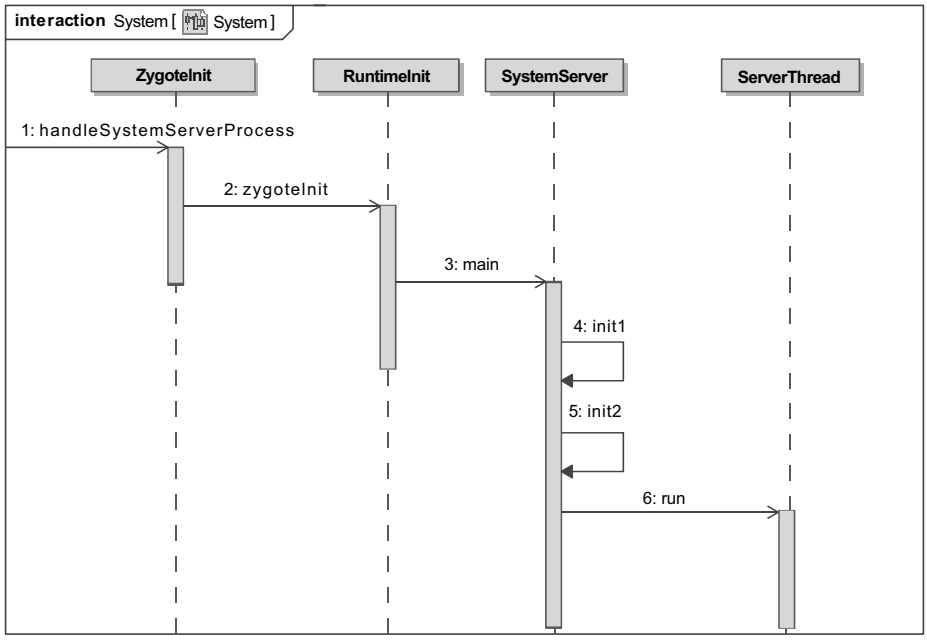


图11-2 System进程的启动过程

这个过程一共分为6个步骤，下面我们就详细分析每一个步骤。

Step 1：ZygoteInit.handleSystemServerProcess

frameworks/base/core/java/com/android/internal/os/ZygoteInit.java

```
01 public class ZygoteInit {
02     ......
03
04     private static void handleSystemServerProcess(
05             ZygoteConnection.Arguments parsedArgs)
06             throws ZygoteInit.MethodAndArgsCaller {
07         closeServerSocket();
08
09         /*
10         * Pass the remaining arguments to SystemServer.
11         * "--nice-name=system_server com.android.server.SystemServer"
12         */
13         RuntimeInit.zygoteInit(parsedArgs.remainingArgs);
14         /* should never reach here */
15     }
16
17     ......
18 }
```

由于System进程复制了Zygote进程的地址空间，因此，它就会获得Zygote进程在启动过程中所创建的一个Socket。System进程不需要使用这个Socket，因此，第7行代码就会调用ZygoteInit类的静态成员函数closeServerSocket来关闭它。接下来第13行代码调用RuntimeInit类的静态成员函数zygoteInit来进一步启动System进程。

Step 2：RuntimeInit.zygoteInit

frameworks/base/core/java/com/android/internal/os/RuntimeInit.java

```
01 public class RuntimeInit {
02     ......
03
04     public static final void zygoteInit(String[] argv)
05             throws ZygoteInit.MethodAndArgsCaller {
06         ......
07
08         commonInit();
09         zygoteInitNative();
10         ......
12
13         // Remaining arguments are passed to the start class's static main
14         String startClass = argv[curArg++];
15         String[] startArgs = new String[argv.length - curArg];
16
17         System.arraycopy(argv, curArg, startArgs, 0, startArgs.length);
18         invokeStaticMain(startClass, startArgs);
19     }
20
21     ......
22 }
```

第8行代码首先调用静态成员函数commonInit来设置System进程的时区和键盘布局等通用信息，接着第9行代码调用静态成员函数zygoteInitNative在System进程中启动一个Binder线程池。在后面的第12章中分析应用程序进程的启动过程时，我们再分析RuntimeInit类的静态成员函数zygoteInitNative的实现。

从前面11.2小节的Step 5可以知道，System进程的启动参数列表argv的最后一个参数的值为“com.android.server.SystemServer”，表示要以com.android.server.SystemServer类的静态成员函数main作为System进程的入口函数。

由于System进程在进入到com.android.server.SystemServer类的静态成员函数main时，已经执行过很多代码了，因此，严格来说，com.android.server.SystemServer类的静态成员函数main并不是System进程的入口函数。为了使得com.android.server.SystemServer类的静态成员函数main好像是System进程的入口函数一样，RuntimeInit类的静态成员函数invokeStaticMain使用了一种巧妙的方法来调用它。在后面的第12章中分析应用程序进程的启动过程时，我们再分析RuntimeInit类的静态成员函数invokeStaticMain的实现。

Step 3：SystemServer.main

frameworks/base/services/java/com/android/server/SystemServer.java

```
01 public class SystemServer
02 {
03     ......
04
05     native public static void init1(String[] args);
06     ......
07
08     public static void main(String[] args) {
09         ......
10
```

```
11          init1(args);
12
13          ......
14      }
15
16      ......
17  }
```

SystemServer类的静态成员函数main主要是调用静态成员函数init1来启动一些使用C++语言开发的系统服务。SystemServer类的静态成员函数init1是一个JNI方法，它是由C++层中的函数android_server_SystemServer_init1来实现的，接下来我们就分析它的调用过程。

Step 4：SystemServer.init1

frameworks/base/services/jni/com_android_server_SystemServer.cpp

```
1 static void android_server_SystemServer_init1(JNIEnv* env, jobject clazz)
2 {
3     system_init();
4 }
```

第3行代码调用了函数system_init来启动一些使用C++语言开发的系统服务，它的实现如下所示。

frameworks/base/cmds/system_server/library/system_init.cpp

```
01 extern "C" status_t system_init()
02 {
03     LOGI("Entered system_init()");
04
05     sp<ProcessState> proc(ProcessState::self());
06
07     sp<IServiceManager> sm = defaultServiceManager();
08
09     sp<GrimReaper> grim = new GrimReaper();
10     sm->asBinder()->linkToDeath(grim, grim.get(), 0);
11
12     char propBuf[PROPERTY_VALUE_MAX];
13     property_get("system_init.startsurfaceflinger", propBuf, "1");
14     if (strcmp(propBuf, "1") == 0) {
15         // Start the SurfaceFlinger
16         SurfaceFlinger::instantiate();
17     }
18
19     // Start the sensor service
20     SensorService::instantiate();
21
22     ......
23
24     AndroidRuntime* runtime = AndroidRuntime::getRuntime();
25
26     LOGI("System server: starting Android services.\n");
27     runtime->callStatic("com/android/server/SystemServer", "init2");
28
29     // If running in our own process, just go into the thread
30     // pool.  Otherwise, call the initialization finished
31     // func to let this process continue its initilization.
32     if (proc->supportsProcesses()) {
33         LOGI("System server: entering thread pool.\n");
34         ProcessState::self()->startThreadPool();
35         IPCThreadState::self()->joinThreadPool();
36         LOGI("System server: exiting thread pool.\n");
37     }
38
39     return NO_ERROR;
40 }
```

第9行代码创建了一个类型为GrimReaper的死亡通知接收者，用来接收Service Manager的死亡通知。

GrimReaper类的定义如下所示。

frameworks/base/cmds/system_server/library/system_init.cpp

```
01 class GrimReaper : public IBinder::DeathRecipient {
02 public:
03     GrimReaper() { }
04
05     virtual void binderDied(const wp<IBinder>& who)
06     {
07         LOGI("Grim Reaper killing system_server...");
08         kill(getpid(), SIGKILL);
09     }
10 };
```

当Service Manager死亡之后，GrimReaper类的成员函数binderDied就会被调用，这时候System进程就会调用函数kill来杀死自己，因为Service Manager不存在了，运行在System进程中的系统服务就无法正常工作了。

回到前面的函数system_init中，接下来第16行和第20行代码分别启动SurfaceFlinger和SensorService两个服务，其中，SurfaceFlinger是一个与系统UI相关的服务，而SensorService是一个传感器服务。接着第27行代码又调用了com.android.server.SystemServer类的静态成员函数init2来启动那些使用Java语言开发的系统服务。最后第34行和第35行代码首先在System进程中启动一个Binder线程池，然后再让当前线程加入到这个Binder线程池中。由于前面的Step 2已经在System进程中启动过Binder线程池了，因此，第34行代码实际上是不起作用的。

接下来，我们继续分析SystemServer类的静态成员函数init2的实现，以便可以看到那些使用Java语言开发的系统服务的启动过程。

Step 5：SystemServer.init2

frameworks/base/services/java/com/android/server/SystemServer.java

```
01 public class SystemServer
02 {
03     ......
04
05     public static final void init2() {
06         Slog.i(TAG, "Entered the Android system server!");
07         Thread thr = new ServerThread();
08         thr.setName("android.server.ServerThread");
09         thr.start();
10     }
11 }
```

第7行代码首先创建了一个类型为ServerThread的线程，然后第9行代码将它启动起来。这个ServerThread线程的入口函数为ServerThread类的成员函数run，接下来我们就分析它的实现。

Step 6：ServerThread.run

frameworks/base/services/java/com/android/server/SystemServer.java

```
01 class ServerThread extends Thread {
02     ......
03
04     @Override
05     public void run() {
06         ......
07         Looper.prepare();
08
09         ......
```

```
10
11          // Critical services...
12          try {
13              ......
14
15              Slog.i(TAG, "Activity Manager");
16              context = ActivityManagerService.main(factoryTest);
17              ......
18
19              Slog.i(TAG, "Package Manager");
20              pm = PackageManagerService.main(context,
21                      factoryTest != SystemServer.FACTORY_TEST_OFF);
22              ActivityManagerService.setSystemProcess();
23              ......
24
25              Slog.i(TAG, "Content Manager");
26              ContentService.main(context,
27                      factoryTest == SystemServer.FACTORY_TEST_LOW_LEVEL);
28              ......
29
30              Slog.i(TAG, "Window Manager");
31              wm = WindowManagerService.main(context, power,
32                      factoryTest != SystemServer.FACTORY_TEST_LOW_LEVE);
33              ServiceManager.addService(Context.WINDOW_SERVICE, wm);
34
35              ......
36          } catch (RuntimeException e) {
37              Slog.e("System", "Failure starting core service", e);
38          }
39          ......
40
41          Looper.loop();
42          Slog.d(TAG, "System ServerThread is exiting!");
43      }
44
45      ......
46  }
```

第7行代码首先调用Looper类的静态成员函数prepare来创建一个消息循环，然后第12行到第38行代码将系统中的关键服务启动起来，并且将它们注册到Service Manager中，最后第41行代码调用Looper类的静态成员函数loop使得当前线程进入到一个消息循环中。在Android系统中，每一个线程都可以有一个消息循环，在后面的第13章中，我们再分析Android系统的消息处理机制。

在前面的第7章到第10章中，我们经常碰到Activity管理服务ActivityManagerService和Content管理服务ContentService；在后面的第14章和第16章中，我们还会使用到Window管理服务WindowManagerService和Package管理服务PackageManagerService，因此，接下来我们就主要分析它们的启动过程。

Activity管理服务ActivityManagerService是由ActivityManagerService类的静态成员函数main启动的，它的实现如下所示。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02          implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03      ......
04
05      static ActivityManagerService mSelf;
06      ......
07
08      public static final Context main(int factoryTest) {
09          AThread thr = new AThread();
10          thr.start();
```

```
11
12          synchronized (thr) {
13              while (thr.mService == null) {
14                  try {
15                      thr.wait();
16                  } catch (InterruptedException e) {
17                  }
18              }
19          }
20
21          ActivityManagerService m = thr.mService;
22          mSelf = m;
23          ......
24
25      }
26
27      static class AThread extends Thread {
28          ActivityManagerService mService;
29          ......
30
31          public void run() {
32              Looper.prepare();
33              ......
34
35              ActivityManagerService m = new ActivityManagerService();
36
37              synchronized (this) {
38                  mService = m;
39                  notifyAll();
40              }
41              ......
42
43              Looper.loop();
44          }
45      }
46
47      ......
48  }
```

第9行和第10行代码首先创建了一个类型为AThread的线程，接着再在这个线程中创建一个ActivityManagerService实例，用来作为系统中的Activity管理服务。

注意

这时候Activity管理服务还没有注册到Service Manager中。等到后面Package管理服务PackageManagerService启动起来之后，前面所创建的ServerThread线程才会将它注册到Service Manager中。

Pakcage管理服务PackageManagerService是由PackageManagerService类的静态成员函数main启动的，它的实现如下所示。

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02      ......
03
04      public static final IPackageManager main(Context context, boolean factoryTest) {
05          PackageManagerService m = new PackageManagerService(context, factoryTest);
06          ServiceManager.addService("package", m);
07          return m;
08      }
09
10      ......
11  }
```

第5行代码首先创建了一个PackageManagerService实例，用来作为系统的Package管理服务；接着第6行代码将它注册到Service Manager中，并且指定它的服务名称为“package”。

Package管理服务PackageManagerService启动起来之后，前面所创建的ServerThread线程就会调用ActivityManagerService类的静态成员函数setSystemProcess将Activity管理服务注册到Service Manager中，它的实现如下所示。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     public static void setSystemProcess() {
06         try {
07             ActivityManagerService m = mSelf;
08
09             ServiceManager.addService("activity", m);
10             ......
11         } catch (PackageManager.NameNotFoundException e) {
12             throw new RuntimeException(
13                     "Unable to find android system package", e);
14         }
15     }
16
17     ......
18 }
```

从这里就可以看出，Activity管理服务ActivityManagerService注册到Service Manager中的名称为“activity”，因此，其他进程就可以通过这个名称来获得它的一个Binder代理对象，从而可以访问它所提供的服务。

Content管理服务ContentService是由ContentService类的静态成员函数main启动的，它的实现如下所示。

frameworks/base/core/java/android/content/ContentService.java

```
01 public final class ContentService extends IContentService.Stub {
02     ......
03
04     public static IContentService main(Context context, boolean factoryTest) {
05         ContentService service = new ContentService(context, factoryTest);
06         ServiceManager.addService(ContentResolver.CONTENT_SERVICE_NAME, service);
07         return service;
08     }
09
10     ......
11 }
```

第5行代码首先创建了一个ContentService实例，用来作为系统的Content管理服务；接着第6行代码将它注册到Service Manager中，并且指定它的服务名称为ContentResolver.CONTENT_SERVICE_NAME，即“content”。

Window管理服务WindowManagerService是由WindowManagerService类的静态成员函数main启动的，它的实现如下所示。

frameworks/base/services/java/com/android/server/WindowManagerService.java

```
01 public class WindowManagerService extends IWindowManager.Stub
02         implements Watchdog.Monitor {
03     ......
04
05     public static WindowManagerService main(Context context,
```

```
06              PowerManagerService pm, boolean haveInputMethods) {
07          WMThread thr = new WMThread(context, pm, haveInputMethods);
08          thr.start();
09
10          synchronized (thr) {
11              while (thr.mService == null) {
12                  try {
13                      thr.wait();
14                  } catch (InterruptedException e) {
15                  }
16              }
17              return thr.mService;
18          }
19      }
20
21      static class WMThread extends Thread {
22          WindowManagerService mService;
23          ......
24
25          public void run() {
26              Looper.prepare();
27              WindowManagerService s = new WindowManagerService(mContext, mPM,
28                      mHaveInputMethods);
29              ......
30
31              synchronized (this) {
32                  mService = s;
33                  notifyAll();
34              }
35
36              Looper.loop();
37          }
38      }
39
40      ......
41  }
```

第7行和第8行代码首先创建了一个类型为WMThread的线程，接着再在这个线程中创建一个WindowManagerService实例，用来作为系统的Window管理服务。这个函数执行完成之后，会返回到ServerThread类的成员函数run中，接着前面所创建的ServerThread线程就会将这个Window管理服务注册到Service Manager中，并且指定它的服务名称为Context.WINDOW_SERVICE，即“window”。

这一步执行完成之后，System进程就将系统的关键服务启动起来了，这时候其他进程就可以通过Service Manager来获得这些服务的代理对象，并且可以通过这些代理对象来使用系统提供的基础服务了。

第 12 章

Android应用程序进程的启动过程

从前面第7章到第10章的内容可以知道，Activity管理服务ActivityManagerService在启动一个应用程序组件时，如果发现这个组件所需要的应用程序进程还没有启动起来，那么它就会请求Zygote进程将这个应用程序进程启动起来。

在前面的第11章中又提到，Zygote进程是通过复制自身的方式来创建一个新的应用程序进程的。由于Zygote进程在启动时会在内部创建一个虚拟机实例，因此，通过复制它而得到的应用程序进程就很自然地获得了一个虚拟机实例拷贝。有了这个虚拟机实例之后，这个应用程序进程就可以将使用Java语言开发的应用程序组件运行起来了。

应用程序进程在启动的过程中，除了可以获得一个虚拟机实例之外，还可以获得一个Binder线程池和一个消息循环，这样，运行在它里面的应用程序组件就可以方便地使用Android系统的消息处理机制，以及Binder进程间通信机制来实现自己的业务逻辑。

接下来，我们首先分析应用程序进程的创建过程，然后分析它的Binder线程池的启动过程，以及消息循环的创建过程。

12.1 应用程序进程的创建过程

每当ActivityManagerService需要创建一个新的应用程序进程来启动一个应用程序组件时，它就会调用ActivityManagerService类的成员函数startProcessLocked向Zygote进程发送一个创建应用程序进程的请求，如图12-1所示。

这个过程一共分为8个步骤，下面我们就详细分析每一个步骤。

Step 1：ActivityManagerService.startProcessLocked

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     private final void startProcessLocked(ProcessRecord app,
06                 String hostingType, String hostingNameStr) {
07         ......
08
09         try {
10             int uid = app.info.uid;
11             int[] gids = null;
12             try {
13                 gids = mContext.getPackageManager().getPackageGids(
```

```
14                      app.info.packageName);
15              } catch (PackageManager.NameNotFoundException e) {
16                  ......
17              }
18              ......
19
20              int pid = Process.start("android.app.ActivityThread",
21                  mSimpleProcessManagement ? app.processName : null, uid, uid,
22                  gids, debugFlags, null);
23
24              ......
25          } catch (RuntimeException e) {
26              ......
27          }
28      }
29
30      ......
31  }
```

第10行到第14行代码首先获得要创建的应用程序进程的用户ID以及用户组ID，接着第20行代码再调用Process类的静态成员函数start来创建这个应用程序进程，并且指定这个应用程序进程的入口函数为android.app.ActivityThread类的静态成员函数main。

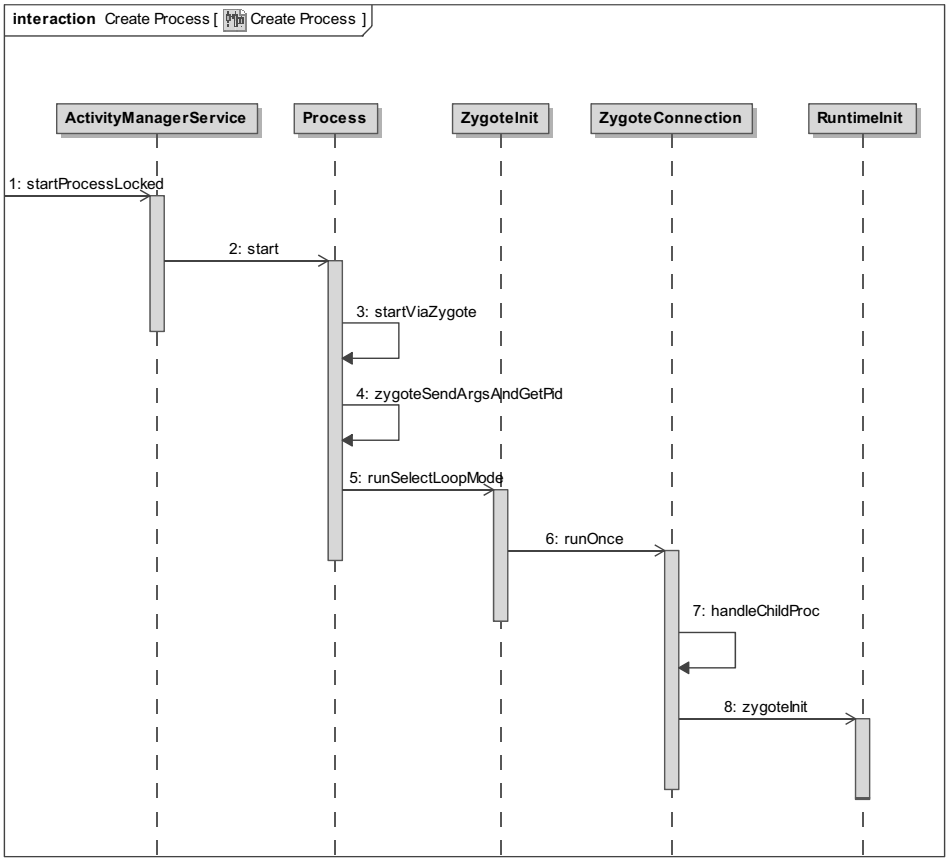


图12–1　应用程序进程的创建过程

Step 2：Process.start

frameworks/base/core/java/android/os/Process.java

```
01 public class Process {
02     ......
03
```

```
04     public static final int start(final String processClass,
05         final String niceName,
06         int uid, int gid, int[] gids,
07         int debugFlags,
08         String[] zygoteArgs)
09     {
10         if (supportsProcesses()) {
11             try {
12                 return startViaZygote(processClass, niceName, uid, gid, gids,
13                     debugFlags, zygoteArgs);
14             } catch (ZygoteStartFailedEx ex) {
15                 ......
16             }
17         } else {
18             // Running in single-process mode
19
20             Runnable runnable = new Runnable() {
21                         public void run() {
22                             Process.invokeStaticMain(processClass);
23                         }
24             };
25
26             // Thread constructors must not be called with null names (see spec).
27             if (niceName != null) {
28                 new Thread(runnable, niceName).start();
29             } else {
30                 new Thread(runnable).start();
31             }
32
33             return 0;
34         }
35     }
36
37     ......
38 }
```

第10行的if语句调用静态成员函数supportsProcesses来判断系统是否支持Binder进程间通信机制。如果支持，那么接下来第12行和第13行代码就会调用静态成员函数startViaZygote来请求Zygote进程创建一个应用程序进程；否则，第20行到第31行代码就会创建一个线程来模拟一个应用程序进程。

由于Android系统是支持Binder进程间通信机制的，因此，接下来我们就分析Process类的成员函数startViaZygote的实现。

Step 3：Process.startViaZygote

frameworks/base/core/java/android/os/Process.java

```
01 public class Process {
02     ......
03
04     private static int startViaZygote(final String processClass,
05             final String niceName,
06             final int uid, final int gid,
07             final int[] gids,
08             int debugFlags,
09             String[] extraArgs)
10             throws ZygoteStartFailedEx {
11         int pid;
12
13         synchronized(Process.class) {
```

```
14              ArrayList<String> argsForZygote = new ArrayList<String>();
15
16              // --runtime-init, --setuid=, --setgid=,
17              // and --setgroups= must go first
18              argsForZygote.add("--runtime-init");
19              argsForZygote.add("--setuid=" + uid);
20              argsForZygote.add("--setgid=" + gid);
21              ......
22
23              // --setgroups is a comma-separated list
24              if (gids != null && gids.length > 0) {
25                  StringBuilder sb = new StringBuilder();
26                  sb.append("--setgroups=");
27
28                  int sz = gids.length;
29                  for (int i = 0; i < sz; i++) {
30                      if (i != 0) {
31                          sb.append(',');
32                      }
33                      sb.append(gids[i]);
34                  }
35
36                  argsForZygote.add(sb.toString());
37              }
38
39              if (niceName != null) {
40                  argsForZygote.add("--nice-name=" + niceName);
41              }
42
43              argsForZygote.add(processClass);
44
45              if (extraArgs != null) {
46                  for (String arg : extraArgs) {
47                      argsForZygote.add(arg);
48                  }
49              }
50
51              pid = zygoteSendArgsAndGetPid(argsForZygote);
52          }
53          ......
54
55          return pid;
56      }
57
58      ......
59  }
```

第14行到第49行代码首先将要创建的应用程序进程的启动参数保存在一个字符串列表argsForZygote中，接着第51行代码再调用Process类的静态成员函数zygoteSendArgsAndGetPid来请求Zygote进程创建这个应用程序进程。

在应用程序进程的启动参数列表中，有一个重要的参数“--runtime-init”，它表示要在新创建的应用程序进程中初始化运行时库，以及启动一个Binder线程池。

Step 4：Process.zygoteSendArgsAndGetPid

frameworks/base/core/java/android/os/Process.java

```
01 public class Process {
02      ......
03
04      static BufferedWriter sZygoteWriter;
05      ......
06
```

```
07     private static int zygoteSendArgsAndGetPid(ArrayList<String> args)
08             throws ZygoteStartFailedEx {
09         int pid;
10
11         openZygoteSocketIfNeeded();
12
13         try {
14             ......
15
16             sZygoteWriter.write(Integer.toString(args.size()));
17             sZygoteWriter.newLine();
18
19             int sz = args.size();
20             for (int i = 0; i < sz; i++) {
21                 String arg = args.get(i);
22                 if (arg.indexOf('\n') >= 0) {
23                     throw new ZygoteStartFailedEx(
24                         "embedded newlines not allowed");
25                 }
26                 sZygoteWriter.write(arg);
27                 sZygoteWriter.newLine();
28             }
29
30             sZygoteWriter.flush();
31
32             // Should there be a timeout on this?
33             pid = sZygoteInputStream.readInt();
34
35             if (pid < 0) {
36                 throw new ZygoteStartFailedEx("fork() failed");
37             }
38         } catch (IOException ex) {
39             ......
40         }
41
42         return pid;
43     }
44
45     ......
46 }
```

第11行代码首先调用静态成员函数openZygoteSocketIfNeeded来创建一个连接到Zygote进程的LocalSocket对象，接着第16行到第30行代码再将要创建的应用程序进程的启动参数列表写入到这个LocalSocket对象中，以便可以将它们传递给Zygote进程。Zygote进程接收到这些数据之后，就会创建一个新的应用程序进程，并且将这个新创建的应用程序进程的PID返回给Activity管理服务ActivityManagerService。

Process类的静态成员函数openZygoteSocketIfNeeded的实现如下所示。

frameworks/base/core/java/android/os/Process.java

```
01 public class Process {
02     ......
03
04     private static final String ZYGOTE_SOCKET = "zygote";
05     ......
06
07     static LocalSocket sZygoteSocket;
08     static DataInputStream sZygoteInputStream;
09     static boolean sPreviousZygoteOpenFailed;
10     ......
11
12     private static void openZygoteSocketIfNeeded()
```

```
13              throws ZygoteStartFailedEx {
14
15          int retryCount;
16
17          if (sPreviousZygoteOpenFailed) {
18              ......
19              retryCount = 0;
20          } else {
21              retryCount = 10;
22          }
23          ......
24
25          for (int retry = 0
26              ; (sZygoteSocket == null) && (retry < (retryCount + 1))
27              ; retry++ ) {
28
29                  if (retry > 0) {
30                      try {
31                          Log.i("Zygote", "Zygote not up yet, sleeping...");
32                          Thread.sleep(ZYGOTE_RETRY_MILLIS);
33                      } catch (InterruptedException ex) {
34                          // should never happen
35                      }
36                  }
37
38                  try {
39                      sZygoteSocket = new LocalSocket();
40                      sZygoteSocket.connect(new LocalSocketAddress(ZYGOTE_SOCKET,
41                          LocalSocketAddress.Namespace.RESERVED));
42
43                      sZygoteInputStream
44                          = new DataInputStream(sZygoteSocket.getInputStream());
45
46                      sZygoteWriter =
47                          new BufferedWriter(
48                          new OutputStreamWriter(
49                          sZygoteSocket.getOutputStream()),
50                          256);
51
52                      Log.i("Zygote", "Process: zygote socket opened");
53
54                      sPreviousZygoteOpenFailed = false;
55                      break;
56                  } catch (IOException ex) {
57                      ......
58                      sZygoteSocket = null;
59                  }
60          }
61          ......
62
63          if (sZygoteSocket == null) {
64              sPreviousZygoteOpenFailed = true;
65              throw new ZygoteStartFailedEx("connect failed");
66          }
67      }
68
69      ......
70  }
```

Process类有一个类型为LocalSocket的静态成员变量sZygoteSocket，它是用来与Zygote进程中一个名称为“zygote”的Socket建立连接的。在前面的11.1和11.2小节中提到，Zygote进程在启动时，会在内部创建一个名称为“zygote”的Socket，这个Socket是与设备文件/dev/socket/zygote绑定在一起的。

第39行代码首先创建一个LocalSocket对象，并且保存在Process类的静态成员变量sZygoteSocket

中；接着第40行和第41行代码将这个LocalSocket对象与一个名称为ZYGOTE_SOCKET，即“zygote”的Socket地址建立连接。在连接的过程中，LocalSocket对象sZygoteSocket会在/dev/socket目录下找到一个对应的zygote文件，然后将它与自己绑定起来，这就相当于与Zygote进程中名称为“zygote”的Socket建立了连接。

连接成功之后，第43行和第44行首先获得LocalSocket对象sZygoteSocket的一个输入流，并且保存在Process类的静态成员变量sZygoteInputStream中，以便获得Zygote进程发送过来的通信数据；接着第46行到第50行代码又获得LocalSocket对象sZygoteSocket的一个输出流，并且保存在Process类的静态成员变量sZygoteWriter中，以便可以向Zygote进程发送通信数据。

在前面的11.2小节中提到，Zygote进程在启动完成之后，会在一个名称为“zygote”的Socket上等待Activity管理服务ActivityManagerService向它发送创建新的应用程序进程的请求。因此，这一步执行完成之后，Zygote进程就会在ZygoteInit类的静态成员函数runSelectLoopMode中接收到一个创建新的应用程序进程的请求。接下来，我们就继续分析它的实现。

Step 5：ZygoteInit.runSelectLoopMode

frameworks/base/core/java/com/android/internal/os/ZygoteInit.java

```
01 public class ZygoteInit {
02     ......
03
04     private static void runSelectLoopMode() throws MethodAndArgsCaller {
05         ArrayList<FileDescriptor> fds = new ArrayList();
06         ArrayList<ZygoteConnection> peers = new ArrayList();
07         FileDescriptor[] fdArray = new FileDescriptor[4];
08
09         fds.add(sServerSocket.getFileDescriptor());
10         peers.add(null);
11         ......
12
13         while (true) {
14             int index;
15             ......
16
17             try {
18                 fdArray = fds.toArray(fdArray);
19                 index = selectReadable(fdArray);
20             } catch (IOException ex) {
21                 throw new RuntimeException("Error in select()", ex);
22             }
23
24             if (index < 0) {
25                 throw new RuntimeException("Error in select()");
26             } else if (index == 0) {
27                 ZygoteConnection newPeer = acceptCommandPeer();
28                 peers.add(newPeer);
29                 fds.add(newPeer.getFileDesciptor());
30             } else {
31                 boolean done;
32                 done = peers.get(index).runOnce();
33
34                 if (done) {
35                     peers.remove(index);
36                     fds.remove(index);
37                 }
38             }
39         }
40     }
41
42     ......
43 }
```

在前面11.2小节的Step 6中提到，当Zygote进程接收到Activity管理服务ActivityManagerService发送过来的一个创建新的应用程序进程的请求之后，就会调用ZygoteConnection类的成员函数runOnce来处理这个请求，如第32行代码所示。

下面我们就分析ZygoteConnection类的成员函数runOnce的实现，以便可以了解它是如何创建和启动一个新的应用程序进程的。

Step 6：ZygoteConnection.runOnce

frameworks/base/core/java/com/android/internal/os/ZygoteConnection.java

```
01 class ZygoteConnection {
02     ......
03
04     private final BufferedReader mSocketReader;
05     ......
06
07     boolean runOnce() throws ZygoteInit.MethodAndArgsCaller {
08         String args[];
09         Arguments parsedArgs = null;
10         ......
11
12         try {
13             args = readArgumentList();
14             ......
15         } catch (IOException ex) {
16             ......
17         }
18         ......
19
20         int pid;
21
22         try {
23             parsedArgs = new Arguments(args);
24
25             ......
26
27             pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid,
28                 parsedArgs.gids, parsedArgs.debugFlags, rlimits);
29         } catch (IllegalArgumentException ex) {
30             ......
31         } catch (ZygoteSecurityException ex) {
32             ......
33         }
34
35         if (pid == 0) {
36             // in child
37             handleChildProc(parsedArgs, descriptors, newStderr);
38             // should never happen
39             return true;
40         }
41         ......
42     }
43
44     ......
45 }
```

第13行代码首先调用成员函数readArgumentList来获得即将要创建的应用程序进程的启动参数，并且保存在一个Arguments对象parsedArgs中；接着第27行和第28行代码调用Zygote类的静态成员函数

forkAndSpecialize来创建这个应用程序进程。

Zygote类的静态成员函数forkAndSpecialize最终是通过函数fork在当前进程中创建一个子进程的，因此，当它的返回值等于0时，就表示是在新创建的子进程中执行的，这时候ZygoteConnection类就会调用成员函数handleChildProc来启动这个子进程。

Step 7：ZygoteConnection.handleChildProc

frameworks/base/core/java/com/android/internal/os/ZygoteConnection.java

```
01 class ZygoteConnection {
02     ......
03
04     private void handleChildProc(Arguments parsedArgs,
05             FileDescriptor[] descriptors, PrintStream newStderr)
06             throws ZygoteInit.MethodAndArgsCaller {
07         ......
08
09         if (parsedArgs.runtimeInit) {
10             RuntimeInit.zygoteInit(parsedArgs.remainingArgs);
11         }
12         ......
13     }
14
15     ......
16 }
```

由于在前面的Step 3中，Activity管理服务ActivityManagerService在新创建的应用程序进程的启动参数列表中设置了一个“--runtime-init”参数，因此，这里传进来的Arguments对象 parsedArgs的成员变量runtimeInit的值就会等于true。接下来第10行就会调用RuntimeInit类的静态成员函数zygoteInit在新创建的应用程序进程中初始化运行时库，以及启动一个Binder线程池。

Step 8：RuntimeInit.zygoteInit

frameworks/base/core/java/com/android/internal/os/RuntimeInit.java

```
01 public class RuntimeInit {
02     ......
03
04     public static final void zygoteInit(String[] argv)
05             throws ZygoteInit.MethodAndArgsCaller {
06         ......
07
08         commonInit();
09         zygoteInitNative();
10         ......
12
13         // Remaining arguments are passed to the start class's static main
14         String startClass = argv[curArg++];
15         String[] startArgs = new String[argv.length - curArg];
16
17         System.arraycopy(argv, curArg, startArgs, 0, startArgs.length);
18         invokeStaticMain(startClass, startArgs);
19     }
20
21     ......
22 }
```

在前面11.3小节的Step 2中，我们已经分析过RuntimeInit类的静态成员函数zygoteInit的实现了。它首先调用静态成员函数commonInit来设置新创建的应用程序进程的时区和键盘布局等通用信息，接着再调用静态成员函数zygoteInitNative在新创建的应用程序进程中启动一个Binder线程池。

从前面的Step 1可以知道，Activity管理服务ActivityManagerService指定了新创建的应用程序进程的

入口函数为ActivityThread类的静态成员函数main。因此，最后第18行实际上是通过调用RuntimeInit类的静态成员函数invokeStaticMain进入到ActivityThread类的静态成员函数main中的。

至此，一个新的应用程序进程就创建完成了。在接下来的12.2和12.3小节中，我们再分析它是如何启动Binder线程池和创建消息循环的。

12.2　Binder线程池的启动过程

从前面12.1小节的内容可以知道，一个新的应用程序进程在创建完成之后，就会调用RuntimeInit类的静态成员函数zygoteInitNative来启动一个Binder线程池，如图12–2所示。

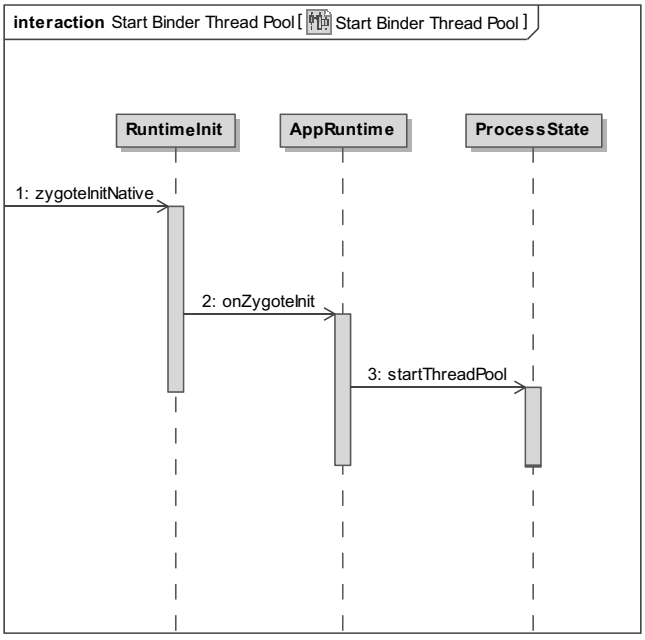


图12–2　Binder线程池的启动过程

这个过程一共分为3个步骤，下面我们就详细分析每一个步骤。

Step 1：RuntimeInit.zygoteInitNative

frameworks/base/core/java/com/android/internal/os/RuntimeInit.java

```
1 public class RuntimeInit {
2     ......
3
4     public static final native void zygoteInitNative();
5
6     ......
7 }
```

RuntimeInit类的静态成员函数zygoteInitNative是一个JNI方法，它是由C++层的函数com_android_internal_os_RuntimeInit_zygoteInit来实现的，如下所示。

frameworks/base/core/jni/AndroidRuntime.cpp

```
1 static AndroidRuntime* gCurRuntime = NULL;
2 ......
3
4 static void com_android_internal_os_RuntimeInit_zygoteInit(JNIEnv* env, jobject clazz)
5 {
6     gCurRuntime->onZygoteInit();
7 }
```

gCurRuntime是一个全局变量，它是在AndroidRuntime类的构造函数中被初始化的，如下所示。

frameworks/base/core/jni/AndroidRuntime.cpp
```
1 AndroidRuntime::AndroidRuntime()
2 {
3     ......
4
5     assert(gCurRuntime == NULL);
6     gCurRuntime = this;
7 }
```

Zygote进程在启动时，会在进程中创建一个AppRuntime对象，如前面11.2小节的Step 1所示。由于AppRuntime类继承了AndroidRuntime类，因此，在创建一个AppRuntime对象时，会导致AndroidRuntime类的构造函数被调用。在调用的过程中，全局变量gCurRuntime就会被初始化，它指向的就是正在创建的AppRuntime对象。又由于每一个新创建的应用进程都复制了Zygote进程的地址空间，因此，在每一个应用程序进程中，都会存在一个全局变量gCurRuntime。

回到函数com_android_internal_os_RuntimeInit_zygoteInit中，第6行代码调用了全局变量gCurRuntime的成员函数onZygoteInit来启动一个Binder线程池。全局变量gCurRuntime指向的是一个AppRuntime对象，并且AppRuntime类重写了父类AndroidRuntime的成员函数onZygoteInit，因此，接下来实际上调用的是AppRuntime类的成员函数onZygoteInit。

Step 2：AppRuntime.onZygoteInit

frameworks/base/cmds/app_process/app_main.cpp
```
01 class AppRuntime : public AndroidRuntime
02 {
03     ......
04
05     virtual void onZygoteInit()
06     {
07         sp<ProcessState> proc = ProcessState::self();
08         if (proc->supportsProcesses()) {
09             LOGV("App process: starting thread pool.\n");
10             proc->startThreadPool();
11         }
12     }
13
14     ......
15 };
```

第8行的if语句判断系统是否支持Binder进程间通信机制。如果支持，那么第10行代码就调用当前应用程序进程中的ProcessState对象的成员函数startThreadPool来启动一个Binder线程池，以便使得当前应用程序进程可以通过Binder进程间通信机制来和其他进程通信。

Step 3：ProcessState.startThreadPool

frameworks/base/libs/binder/ProcessState.cpp
```
1 void ProcessState::startThreadPool()
2 {
3     AutoMutex _l(mLock);
4     if (!mThreadPoolStarted) {
5         mThreadPoolStarted = true;
6         spawnPooledThread(true);
7     }
8 }
```

ProcessState类有一个成员变量mThreadPoolStarted，它的初始值等于false。在前面的5.2小节中提到，在Android系统中，每一个支持Binder进程间通信机制的进程内部都有一个唯一的ProcessState对象。当这个ProcessState对象的成员函数startThreadPool第一次被调用时，它就会在当前进程中启动一

个Binder线程池，并且将它的成员变量mThreadPoolStarted的值设置为true，避免以后重复执行启动Binder线程池的操作。

ProcessState类的成员函数startThreadPool的实现可以参考前面5.8.2小节的内容。当它执行完成之后，新创建的应用程序进程就可以支持Binder进程间通信机制了。这意味着我们以后在应用程序进程中实现自己的Binder本地对象时，只需要将它启动起来并且注册到Service Manager中就可以了，而不必关心它是如何通过Binder线程来接收进程间通信请求的。

至此，应用程序进程的Binder线程池的启动过程就分析完成了。接下来，我们继续分析应用程序进程的消息循环的创建过程。

12.3　消息循环的创建过程

从前面12.1小节的内容可以知道，一个新的应用程序进程在创建完成之后，就会调用RuntimeInit类的静态成员函数invokeStaticMain将ActivityThread类的静态成员函数main设置为新创建的应用程序进程的入口函数。ActivityThread类的静态成员函数main在调用的过程中，就会在当前应用程序进程中创建一个消息循环。

接下来，我们就从RuntimeInit类的静态成员函数invokeStaticMain开始分析应用程序进程的消息循环的创建过程，如图12-3所示。

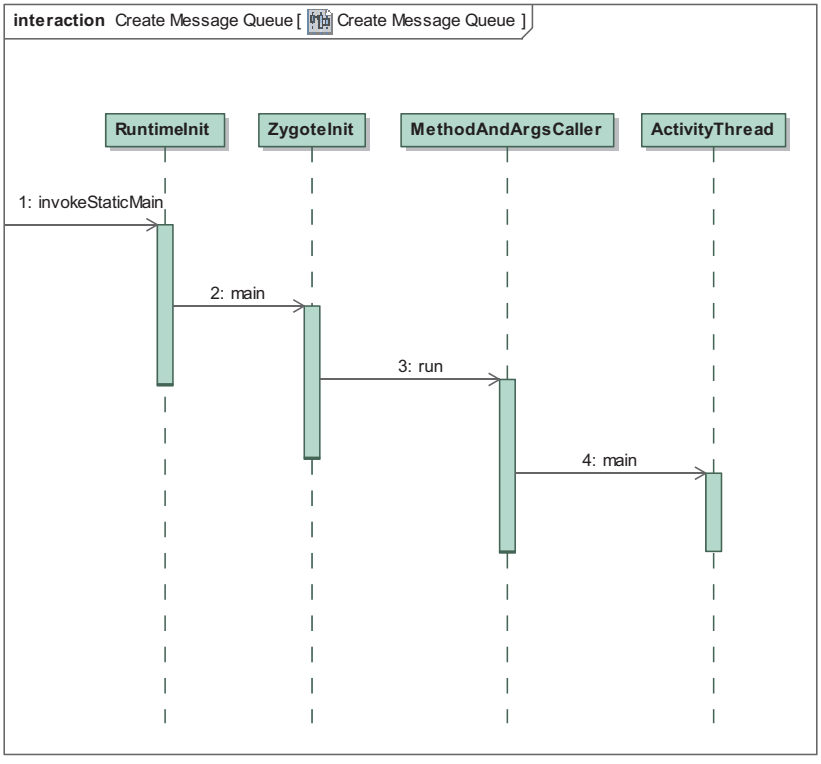


图12-3　消息循环的创建过程

这个过程一共分为4个步骤，下面我们就详细分析每一个步骤。

Step 1：RuntimeInit.invokeStaticMain

frameworks/base/core/java/com/android/internal/os/RuntimeInit.java

```
01 public class RuntimeInit {
02     ......
```

```
03
04      static void invokeStaticMain(ClassLoader loader,
05              String className, String[] argv)
06              throws ZygoteInit.MethodAndArgsCaller {
07          Class<?> cl;
08
09          try {
10              cl = loader.loadClass(className);
11          } catch (ClassNotFoundException ex) {
12              ......
13          }
14
15          Method m;
16          try {
17              m = cl.getMethod("main", new Class[] { String[].class });
18          } catch (NoSuchMethodException ex) {
19              ......
20          } catch (SecurityException ex) {
21              ......
22          }
23          ......
24
25          /*
26          * This throw gets caught in ZygoteInit.main(), which responds
27          * by invoking the exception's run() method. This arrangement
28          * clears up all the stack frames that were required in setting
29          * up the process.
30          */
31          throw new ZygoteInit.MethodAndArgsCaller(m, argv);
32      }
33
34      ......
35  }
```

参数className的值等于“android.app.ActivityThread”，它是由Activity管理服务ActivityManagerService在请求Zygote进程创建一个新的应用程序进程时指定的。

第10行代码首先将android.app.ActivityThread类加载到当前应用程序进程中；接着第17行代码获得它的静态成员函数main，并且保存在一个Method对象中；最后第31行代码将这个Method对象封装在一个MethodAndArgsCaller对象中，并且将这个MethodAndArgsCaller对象作为一个异常对象抛出来给当前应用程序进程处理。

前面提到，新创建的应用程序进程复制了Zygote进程的地址空间，因此，当前新创建的应用程序进程的调用堆栈与Zygote进程的调用堆栈是一致的。从前面11.2小节的内容可以知道，Zygote进程最开始执行的是应用程序app_process的入口函数main，接着再调用ZygoteInit类的静态成员函数main，最后进入到ZygoteInit类的静态成员函数runSelectLoopMode来循环等待Activity管理服务ActivityManagerService向它发送创建新的应用程序进程的请求。当Zygote进程接收到Activity管理服务ActivityManagerService发送过来的创建新的应用程序进程的请求之后，它就会创建一个新的应用程序进程，并且让这个新创建的应用程序进程沿着ZygoteInit类的静态成员函数runSelectLoopMode一直执行到RuntimeInit类的静态成员函数invokeStaticMain。因此，当RuntimeInit类的静态成员函数invokeStaticMain抛出一个类型为MethodAndArgsCaller的异常时，系统就会沿着这个调用过程往后找到一个合适的代码块来捕获它。

由于ZygoteInit类的静态成员函数main捕获了类型为MethodAndArgsCaller的异常，因此，接下来它就会被调用，以便可以处理这里抛出的一个MethodAndArgsCaller异常。

Step 2：ZygoteInit.main

frameworks/base/core/java/com/android/internal/os/RuntimeInit.java

```
01 public class ZygoteInit {
02     ......
03
04     public static void main(String argv[]) {
05         try {
06             ......
07         } catch (MethodAndArgsCaller caller) {
08             caller.run();
09         } catch (RuntimeException ex) {
10             ......
11         }
12     }
13
14     ......
15 }
```

ZygoteInit类的静态成员函数main捕获了一个类型为MethodAndArgsCaller的异常之后，就会调用MethodAndArgsCaller类的成员函数run来进一步处理。

Step 3：MethodAndArgsCaller.run

frameworks/base/core/java/com/android/internal/os/RuntimeInit.java

```
01 public class ZygoteInit {
02     ......
03
04     public static class MethodAndArgsCaller extends Exception
05             implements Runnable {
06         /** method to call */
07         private final Method mMethod;
08
09         /** argument array */
10         private final String[] mArgs;
11
12         public MethodAndArgsCaller(Method method, String[] args) {
13             mMethod = method;
14             mArgs = args;
15         }
16
17         public void run() {
18             try {
19                 mMethod.invoke(null, new Object[] { mArgs });
20             } catch (IllegalAccessException ex) {
21                 ......
22             } catch (InvocationTargetException ex) {
23                 ......
24             }
25         }
26     }
27
28     ......
29 }
```

MethodAndArgsCaller类有两个成员变量mMethod和mArgs，其中，成员变量mMethod指向了ActivityThread类的静态成员函数main，而成员变量mArgs指向了新创建的应用程序进程的启动参数。

MethodAndArgsCaller类的成员函数run的实现很简单，它通过MethodAndArgsCaller类的成员变量mMethod调用了ActivityThread类的静态成员函数main。从这里我们就可以看到，在前面的Step 1中，抛出一个类型为MethodAndArgsCaller的异常的目的就是为了调用ActivityThread类的静态成员函数main。

读者可能会觉得奇怪，绕了一大圈就是为了调用ActivityThread类的静态成员函数main，为什么不直接在前面的Step 1中调用呢？

前面提到，Activity管理服务ActivityManagerService请求Zygote进程创建的应用程序进程的入口函数为ActivityThread类的静态成员函数main。但是由于新创建的应用程序进程一开始就需要在内部初始化运行时库，以及启动Binder线程池，因此，当ActivityThread类的静态成员函数main被调用时，新创建的应用程序进程实际上已经执行了相当多的代码。为了使得新创建的应用程序进程觉得它的入口函数就是ActivityThread类的静态成员函数main，系统就不可以在前面的Step 1中直接调用它，而是先抛出一个异常回到ZygoteInit类的静态成员函数main中，然后再间接地调用它，这样就可以巧妙地利用Java语言的异常处理机制来清理它前面的调用堆栈了。

接下来，我们就继续分析ActivityThread类的静态成员函数main的实现。

Step 4：ActivityThread.main

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     public static final void main(String[] args) {
05         ......
06
07         Looper.prepareMainLooper();
08         ......
09
10         ActivityThread thread = new ActivityThread();
11         thread.attach(false);
12         ......
13
14         Looper.loop();
15
16         ......
17     }
18
19     ......
20 }
```

第7行代码首先调用Looper类的静态成员函数prepareMainLooper在当前应用程序进程中创建一个消息循环，然后第10行和第11行代码在当前应用程序进程中创建一个ActivityThread实例，最后第14行代码调用Looper类的静态成员函数loop进入到前面所创建的一个消息循环中。

至此，新创建的应用程序进程的消息循环就创建完成了，整个应用程序进程的启动过程也完成了。从这里就可以看出，在Android系统中，每一个应用程序进程在启动完成之后，都会自动地进入到一个消息循环中，这样运行在它里面的应用程序组件就可以方便地使用Android系统的消息处理机制来实现自己的业务逻辑。

第13章 Android应用程序的消息处理机制

我们知道，Android应用程序是通过消息来驱动的。Android应用程序的每一个线程在启动时，都可以首先在内部创建一个消息队列，然后再进入到一个无限循环中，不断检查它的消息队列是否有新的消息需要处理。如果有新的消息需要处理，那么线程就会将它从消息队列中取出来，并且对它进行处理；否则，线程就会进入睡眠等待状态，直到有新的消息需要处理为止。这样就可以通过消息来驱动Android应用程序的执行了。

Android应用程序的消息处理机制是围绕消息队列来实现的。一个线程拥有一个消息队列之后，就可以进入到一个消息循环中，同时其他线程以及线程本身可以往这个消息队列发送消息，以便可以在这个消息被处理时执行一个特定的操作。这样我们就将一个线程的生命周期划分为创建消息队列和进入消息循环两个阶段，其中，消息循环阶段又划分为发送消息和处理消息两个子阶段，它们是交替进行的。

Android系统主要通过MessageQueue、Looper和Handler三个类来实现Android应用程序的消息处理机制，其中，MessageQueue类用来描述消息队列；Looper类用来创建消息队列，以及进入消息循环；Handler类用来发送消息和处理消息。接下来，我们首先分析线程消息队列的创建过程，然后分析线程的消息循环过程，最后分析线程消息的发送和处理过程。

13.1 创建线程消息队列

Android应用程序线程的消息队列是使用一个MessageQueue对象来描述的，它可以通过调用Looper类的静态成员函数prepareMainLooper或者prepare来创建，其中，前者用来为应用程序的主线程创建消息队列；而后者用来为应用程序的其他子线程创建消息队列。

在分析Android应用程序线程的消息队列的创建过程之前，我们首先简单介绍一下Looper类和MessageQueue类的实现。

由于Android应用程序的消息处理机制不仅可以在Java代码中使用，也可以在C++代码中使用，因此，Android系统在C++层中也有一个相应的Looper类和NativeMessageQueue类。其中，Java层中的Looper类和MessageQueue类是通过C++层中的Looper类和NativeMessageQueue类来实现的，它们的关系如图13-1所示。

Java层中的每一个Looper对象内部都有一个类型为MessageQueue的成员变量mQueue，它指向了一个MessageQueue对象；而在C++层中，每一个NativeMessageQueue对象内部都有一个类型为Looper的成员变量mLooper，它指向了一个C++层的Looper对象。

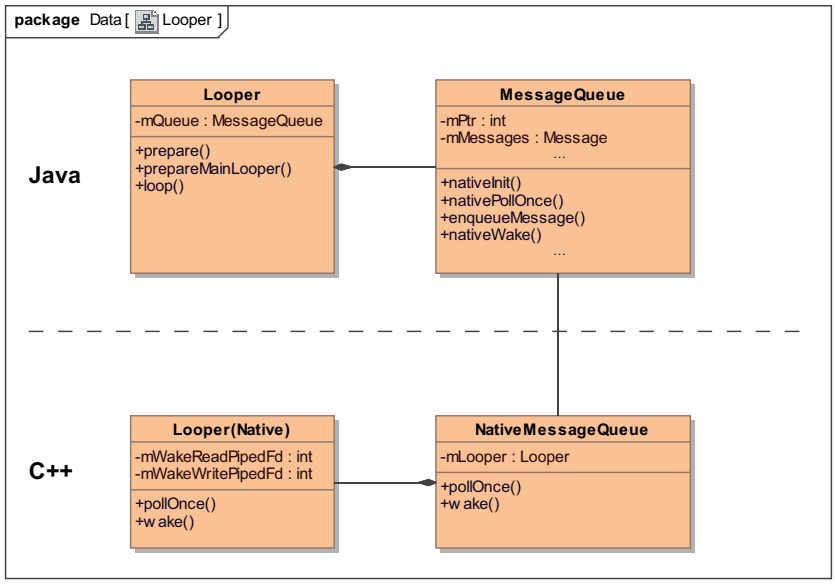


图13-1　Looper类、MessageQueue类、Looper（Native）类和NativeMessageQueue类的关系

Java层中的每一个MessageQueue对象都有一个类型为int的成员变量mPtr，它保存了C++层中的一个NativeMessageQueue对象的地址值，这样我们就可以将Java层中的一个MessageQueue对象与C++层中的一个NativeMessageQueue对象关联起来。

Java层中的每一个MessageQueue对象还有一个类型为Message的成员变量mMessages，它用来描述一个消息队列，我们可以调用MessageQueue类的成员函数enqueueMessage来往里面添加一个消息。

C++层中的Looper对象有两个类型为int的成员变量mWakeReadPipeFd和mWakeWritePipeFd，它们分别用来描述一个管道的读端文件描述符和写端文件描述符。当一个线程的消息队列没有消息需要处理时，它就会在这个管道的读端文件描述符上进行睡眠等待，直到其他线程通过这个管道的写端文件描述符来唤醒它为止。

当我们调用Java层的Looper类的静态成员函数prepareMainLooper或者prepare来为一个线程创建一个消息队列时，Java层的Looper类就会在这个线程中创建一个Looper对象和一个MessageQueue对象。在创建Java层的MessageQueue对象的过程中，又会调用它的成员函数nativeInit在C++层中创建一个NativeMessageQueue对象和一个Looper对象。在创建C++层的Looper对象时，又会创建一个管道，这个管道的读端文件描述符和写端文件描述符就保存在它的成员变量mWakeReadPipeFd和mWakeWritePipeFd中。

Java层中的Looper类的成员函数loop、MessageQueue类的成员函数nativePollOnce和nativeWake，以及C++层中的NativeMessageQueue类的成员函数pollOnce和wake、Looper类的成员函数pollOnce和wake，是与线程的消息循环、消息发送和消息处理相关的，在接下来的13.2、13.3和13.4小节中，我们再详细分析它们的实现。

理解了这些背景知识之后，接下来我们就可以分析Android应用程序线程的消息队列的创建过程了，即分析Looper类的静态成员函数prepareMainLooper和prepare的实现。

Looper类的静态成员函数prepareMainLooper和prepare的实现如下所示。

frameworks/base/core/java/android/os/Looper.java

```
01 public class Looper {
02     ......
03
04     private static final ThreadLocal sThreadLocal = new ThreadLocal();
05     final MessageQueue mQueue;
```

```
06      ......
07
08      private static Looper mMainLooper = null;
09      ......
10
11      public static final void prepare() {
12          if (sThreadLocal.get() != null) {
13              throw new RuntimeException("Only one Looper may be created per thread");
14          }
15          sThreadLocal.set(new Looper());
16      }
17
18      public static final void prepareMainLooper() {
19          prepare();
20          setMainLooper(myLooper());
21          ......
22      }
23
24      private synchronized static void setMainLooper(Looper looper) {
25          mMainLooper = looper;
26      }
27
28      public synchronized static final Looper getMainLooper() {
29          return mMainLooper;
30      }
31
32      public static final Looper myLooper() {
33          return (Looper)sThreadLocal.get();
34      }
35
36      ......
37  }
```

Looper类有一个类型为ThreadLocal的静态成员变量sThreadLocal，它是用来保存线程中的Looper对象的。我们可以将Looper类的静态成员变量sThreadLocal理解为一个线程局部变量，或者一个HashMap，每一个创建了消息队列的Android应用程序线程在里面都有一个关联的Looper对象。当我们调用这个静态成员变量的成员函数get或者Looper类的静态成员函数myLooper时，就可以获得与当前线程关联的一个Looper对象；而当我们调用这个静态成员变量的成员函数set时，就可以将一个Looper对象与当前线程关联起来。

在Looper类的静态成员函数prepare中，第12行的if语句检查当前线程是否已经有一个Looper对象了。如果有，那么第13行代码就会抛出一个异常；否则，第15行代码就会首先创建一个Looper对象，然后将这个Looper对象保存在Looper类的静态成员变量sThreadLocal中。

在Looper类的静态成员函数prepareMainLooper中，第19行代码首先调用Looper类的静态成员函数prepare在当前线程中创建一个Looper对象，接着第20行代码调用Looper类的静态成员函数setMainLooper将这个Looper对象保存在Looper类的静态成员变量mMainLooper中。

注意

Looper类的静态成员函数prepareMainLooper只能在Android应用程序主线程中调用。Android应用程序主线程是一个特殊的线程，因为只有它才可以执行与UI相关的操作，因此，我们又将它称作UI线程。将Android应用程序主线程的Looper对象另外保存在一个独立的静态成员变量中，是为了让其他线程可以通过Looper类的静态成员函数getMainLooper来访问它，从而可以往它的消息队列中发送一些与UI操作相关的消息。

一个Looper对象在创建的过程中，会在内部创建一个MessageQueue对象，并且保存在它的成员变

量mQueue中，如下所示。

frameworks/base/core/java/android/os/Looper.java

```
01 public class Looper {
02     ......
03
04     private Looper() {
05         mQueue = new MessageQueue();
06         ......
07     }
08
09     ......
10 }
```

一个MessageQueue对象在创建的过程中，又会在C++层中创建一个NativeMessageQueue对象，这是通过调用MessageQueue类的成员函数nativeInit来实现的，如下所示。

frameworks/base/core/java/android/os/MessageQueue.java

```
01 public class MessageQueue {
02     ......
03
04     private int mPtr; // used by native code
05
06     private native void nativeInit();
07
08     MessageQueue() {
09         nativeInit();
10     }
11
12     ......
13 }
```

MessageQueue类的成员函数nativeInit是一个JNI方法，它是由C++层中的函数android_os_MessageQueue_nativeInit来实现的，如下所示。

frameworks/base/core/jni/android_os_MessageQueue.cpp

```
1 static void android_os_MessageQueue_nativeInit(JNIEnv* env, jobject obj) {
2     NativeMessageQueue* nativeMessageQueue = new NativeMessageQueue();
3     ......
4
5     android_os_MessageQueue_setNativeMessageQueue(env, obj, nativeMessageQueue);
6 }
```

第2行代码首先在C++层中创建一个NativeMessageQueue对象，接着第5行代码调用函数android_os_MessageQueue_setNativeMessageQueue将它与参数obj所描述的一个Java层的MessageQueue对象关联起来。

函数android_os_MessageQueue_setNativeMessageQueue的实现如下所示。

```
frameworks/base/core/jni/android_os_MessageQueue.cpp
1 static void android_os_MessageQueue_setNativeMessageQueue(JNIEnv* env, jobject messageQueueObj,
2         NativeMessageQueue* nativeMessageQueue) {
3     env->SetIntField(messageQueueObj, gMessageQueueClassInfo.mPtr,
4             reinterpret_cast<jint>(nativeMessageQueue));
5 }
```

gMessageQueueClassInfo是一个全局变量，它的定义如下所示。

frameworks/base/core/jni/android_os_MessageQueue.cpp

```
1 static struct {
2     jclass clazz;
3
4     jfieldID mPtr;   // native object attached to the DVM MessageQueue
5 } gMessageQueueClassInfo;
```

它的成员变量clazz指向了Java层的MessageQueue类；成员变量mPtr指向了MessageQueue类的成员变量mPtr。

从全局变量gMessageQueueClassInfo的定义就可以看出，函数android_os_MessageQueue_setNativeMessageQueue实际上是将一个C++层的NativeMessageQueue对象的地址值保存在一个Java层的MessageQueue对象的成员变量mPtr中，从而可以将它们关联起来。

一个NativeMessageQueue对象在创建的过程中，又会在内部创建一个C++层的Looper对象，如下所示。

frameworks/base/core/jni/android_os_MessageQueue.cpp

```
1 NativeMessageQueue::NativeMessageQueue() {
2     mLooper = Looper::getForThread();
3     if (mLooper == NULL) {
4         mLooper = new Looper(false);
5         Looper::setForThread(mLooper);
6     }
7 }
```

第2行代码首先调用C++层的Looper类的静态成员函数getForThread来检查是否已经为当前线程创建过一个C++层的 Looper对象。如果没有创建过，那么接着第4行代码首先创建一个C++层的Looper对象，并且将它保存在NativeMessageQueue类的成员变量mLooper中，然后第5行代码再调用C++层的Looper类的静态成员函数setForThread将它与当前线程关联起来。

一个C++层的Looper对象在创建的过程中，又会在内部创建一个管道，如下所示。

frameworks/base/libs/utils/Looper.cpp

```
01 Looper::Looper(bool allowNonCallbacks) :
02     mAllowNonCallbacks(allowNonCallbacks),
03     mResponseIndex(0) {
04     int wakeFds[2];
05     int result = pipe(wakeFds);
06     ......
07
08     mWakeReadPipeFd = wakeFds[0];
09     mWakeWritePipeFd = wakeFds[1];
10
11     ......
12
13     // Allocate the epoll instance and register the wake pipe.
14     mEpollFd = epoll_create(EPOLL_SIZE_HINT);
15     ......
16
17     struct epoll_event eventItem;
18     memset(& eventItem, 0, sizeof(epoll_event)); // zero out unused members of data field union
19     eventItem.events = EPOLLIN;
20     eventItem.data.fd = mWakeReadPipeFd;
21     result = epoll_ctl(mEpollFd, EPOLL_CTL_ADD, mWakeReadPipeFd, & eventItem);
22     ......
23 }
```

第5行代码首先调用函数pipe创建了一个管道，接着第8行和第9行代码分别将这个管道的读端文件描述符和写端文件描述符保存在C++层的Looper类的成员变量mWakeReadPipeFd和mWakeWritePipeFd中。

这个管道在一个线程的消息循环过程中起到的作用非常大。首先，当一个线程没有新的消息需要处理时，它就会睡眠在这个管道的读端文件描述符上，直到有新的消息需要处理为止；其次，当其他线程向这个线程的消息队列发送了一个消息之后，其他线程就会通过这个管道的写端文件描述符往这个管道

写入一个数据，从而将这个线程唤醒，以便它可以对刚才发送到它的消息队列中的消息进行处理。

第14行代码首先调用函数epoll_create来创建一个epoll实例，并且将它的文件描述符保存在C++层的Looper类的成员变量mEpollFd中。接着第17行到第21行代码将前面所创建的管道的读端文件描述符添加到这个epoll实例中，以便可以对它所描述的管道的写操作进行监听。

Linux系统的epoll机制是为了同时监听多个文件描述符的IO读写事件而设计的，它是一个多路复用IO接口，类似于Linux系统的select机制，但是它是select机制的增强版。如果一个epoll实例监听了大量的文件描述符的IO读写事件，但是只有少量的文件描述符是活跃的，即只有少量的文件描述符会发生IO读写事件，那么这个epoll实例可以显著地减少CPU的使用率，从而提高系统的并发处理能力。

读者可能会觉得奇怪，前面所创建的epoll实例只监听了一个文件描述符的IO写事件，即前面所创建的管道的IO写事件，这值得使用epoll机制来实现吗？其实，以后我们还可以调用C++层的Looper类的成员函数addFd向这个epoll实例中注册更多的文件描述符，以便可以监听它们的IO读写事件，这样就可以充分地利用一个线程的消息循环来做其他事情了。在后面的第14章中分析Android应用程序的键盘消息处理机制时，我们就会看到C++层的Looper类的成员函数addFd的使用场景。

至此，一个Android应用程序线程的消息队列就创建完成了。接下来，我们再分析这个Android应用程序线程是如何围绕这个消息队列来进行消息循环的。

13.2 线程消息循环过程

一个Android应用程序线程的消息队列创建完成之后，我们就可以调用Looper类的静态成员函数loop使它进入到一个消息循环中，如图13-2所示。

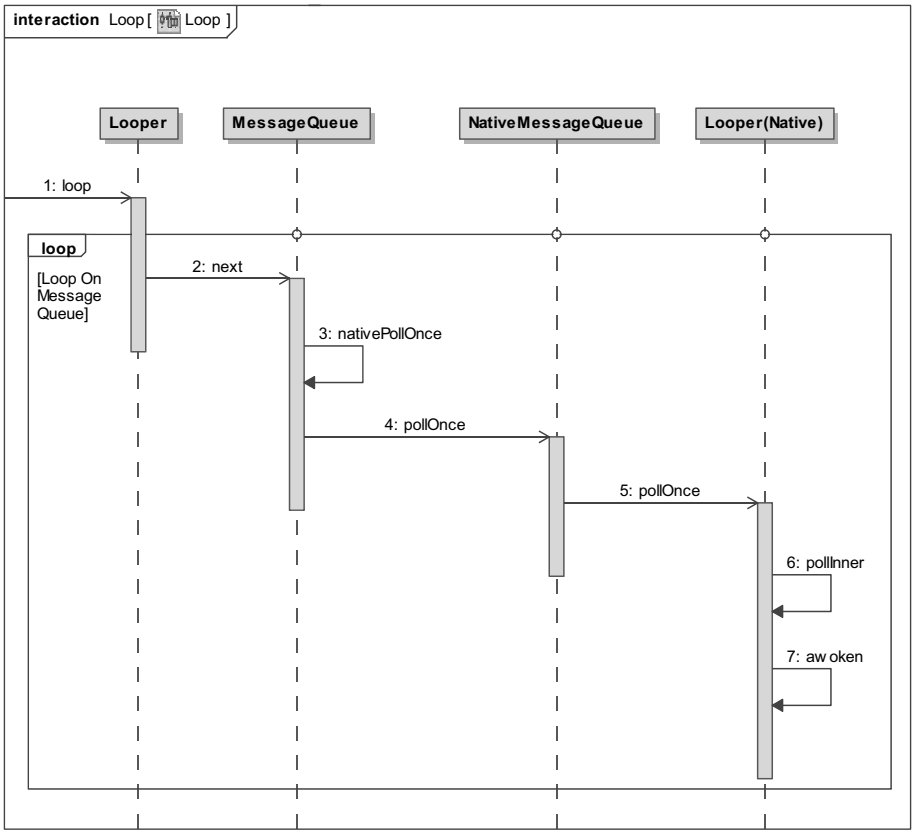


图13-2 线程消息循环过程

这个过程一共分为7个步骤，下面我们就详细分析每一个步骤。

Step 1：Looper.loop

frameworks/base/core/java/android/os/Looper.java

```
01 public class Looper {
02     ......
03
04     public static final void loop() {
05         Looper me = myLooper();
06         MessageQueue queue = me.mQueue;
07         ......
08
09         while (true) {
10             Message msg = queue.next(); // might block
11             ......
12
13             if (msg != null) {
14                 ......
15             }
16         }
17     }
18
19     ......
20 }
```

第5行和第6行代码首先获得当前线程的消息队列，接着第9行到第16行的while循环不断地检查这个消息队列中是否有新的消息需要处理。如果有新的消息需要处理，那么第10行代码获得的Message对象msg就不等于null；否则，当前线程就会在MessageQueue类的成员函数next中进入睡眠等待状态，直到有新的消息需要处理为止。

Step 2：MessageQueue.next

frameworks/base/core/java/android/os/MessageQueue.java

```
01 public class MessageQueue {
02     ......
03
04     final Message next() {
05         int pendingIdleHandlerCount = -1; // -1 only during first iteration
06         int nextPollTimeoutMillis = 0;
07
08         for (;;) {
09             if (nextPollTimeoutMillis != 0) {
10                 Binder.flushPendingCommands();
11             }
12             nativePollOnce(mPtr, nextPollTimeoutMillis);
13
14             synchronized (this) {
15                 // Try to retrieve the next message.  Return if found.
16                 final long now = SystemClock.uptimeMillis();
17                 final Message msg = mMessages;
18                 if (msg != null) {
19                     final long when = msg.when;
20                     if (now >= when) {
21                         ......
22                         mMessages = msg.next;
23                         msg.next = null;
24                         ......
25                         return msg;
26                     } else {
27                         nextPollTimeoutMillis = (int) Math.min(when - now, Integer.MAX_VALUE);
28                     }
```

```
29                  } else {
30                      nextPollTimeoutMillis = -1;
31                  }
32                  ......
33
34                  // Get the idle handlers
35                  ......
36              }
37
38
39              // Run the idle handlers
40              ......
41
42              // While calling an idle handler, a new message could have been delivered
43              // so go back and look again for a pending message without waiting.
44              nextPollTimeoutMillis = 0;
45          }
46      }
47
48      ......
49  }
```

第5行代码定义了一个变量pendingIdleHandlerCount，用来保存注册到消息队列中的空闲消息处理器（IdleHandler）的个数，它们是在第35行省略的代码中计算的。当线程发现它的消息队列没有新的消息需要处理时，不是马上就进入睡眠等待状态，而是先调用注册到它的消息队列中的IdleHandler对象的成员函数queueIdle，以便它们有机会在线程空闲时执行一些操作。这些IdleHandler对象的成员函数queueIdle是在第40行省略的代码中调用的，在后面的13.4小节中，我们再分析它们的调用过程。

第6行代码定义了另外一个变量nextPollTimeoutMillis，用来描述当消息队列中没有新的消息需要处理时，当前线程需要进入睡眠等待状态的时间。如果变量nextPollTimeoutMillis的值等于0，那么就表示即使消息队列中没有新的消息需要处理，当前线程也不要进入睡眠等待状态。如果变量nextPollTimeoutMillis的值等于-1，那么就表示当消息队列中没有新的消息需要处理时，当前线程需要无限地处于睡眠等待状态，直到它被其他线程唤醒为止。

第8行到第45行的for循环不断地调用成员函数nativePollOnce来检查当前线程的消息队列中是否有新的消息需要处理。

注意

在调用成员函数nativePollOnce时，当前线程可能会进入睡眠等待状态。如果需要进入睡眠等待状态，那么进入睡眠等待状态的时间就由第二个参数nextPollTimeoutMillis来指定。

MessageQueue类内部有一个类型为Message的成员变量mMessages，用来描述当前线程需要处理的消息。当前线程从成员函数nativePollOnce返回来之后，如果它有新的消息需要处理，即MessageQueue类的成员变量mMessages不等于null，那么接下来第19行到28行代码就会对它进行处理；否则，第30行代码就会将变量nextPollTimeoutMillis的值设置为-1，表示当前线程下次在调用成员函数nativePollOnce时，如果没有新的消息需要处理，那么就要无限地处于睡眠等待状态，直到它被其他线程唤醒为止。

第19行代码首先获得MessageQueue类的成员变量mMessages所描述的消息的处理时间。如果这个消息的处理时间小于等于系统的当前时间，那么就说明当前线程需要马上对它进行处理。因此，接下来第21行到第25行代码就会将它返回给上一步处理，并且将MessageQueue类的成员变量mMessages重新指向下一个需要处理的消息；否则，第27行代码就会计算当前线程下一次调用成员函数nativePollOnce时，如果没有新的消息需要处理，那么当前线程需要进入睡眠等待状态的时间，以便可以在指定的时间点对接下来的消息进行处理。在接下来的13.3小节中，我们就会看到，每一个发送到消息队列中的消息都

有一个指定的处理时间，而这些消息就是按照它们的处理时间从小到大的顺序排列在消息队列中的。

当前线程执行到第40行代码时，就说明它目前没有新的消息需要处理，于是接下来它就会重新执行第8行到第45行的for循环来等待下一个需要处理的消息。这时候当前线程需要进入睡眠等待状态的时间就保存在变量nextPollTimeoutMillis中。不过，在进入睡眠等待状态之前，当前线程会分发一个线程空闲消息给那些已经注册到消息队列中的IdleHandler对象来处理，如第40行省略的代码所示。

第44行代码是在注册到消息队列中的IdleHandler对象处理完成一个线程空闲消息之后执行的，它主要是将变量nextPollTimeoutMillis的值重新设置为0，表示当前线程接下来在调用成员函数nativePollOnce时，不可以进入睡眠等待状态。由于注册的IdleHandler对象在处理线程空闲消息期间，其他线程可能已经向当前线程的消息队列发送了一个或者若干个消息，因此，这时候当前线程就不能够在成员函数nativePollOnce中进入睡眠等待状态，而是要马上返回来检查它的消息队列中是否有新的消息需要处理。

注意

在第8行到第45行的for循环中，每一次调用成员函数nativePollOnce之前，第9行的if语句都会检查变量nextPollTimeoutMillis的值是否等于0。如果不等于0，那么就说明当前线程会在成员函数nativePollOnce中进入睡眠等待状态。这时候第10行代码就会调用Binder类的静态成员函数flushPendingCommands来处理那些正在等待处理的Binder进程间通信请求，避免它们长时间得不到处理。

接下来，我们就继续分析MessageQueue类的成员函数nativePollOnce的实现，以便可以了解当前线程是如何检查自己是否有新的消息需要处理的。

Step 3：MessageQueue.nativePollOnce

```
frameworks/base/core/jni/android_os_MessageQueue.cpp
1 static void android_os_MessageQueue_nativePollOnce(JNIEnv* env, jobject obj,
2         jint ptr, jint timeoutMillis) {
3     NativeMessageQueue* nativeMessageQueue = reinterpret_cast<NativeMessageQueue*>(ptr);
4     nativeMessageQueue->pollOnce(timeoutMillis);
5 }
```

MessageQueue类的成员函数nativePollOnce是一个JNI方法，它是由C++层中的函数android_os_MessageQueue_nativePollOnce来实现的。

参数obj指向了一个Java层的MessageQueue对象，而参数ptr指向了这个MessageQueue对象的成员变量mPtr。从前面13.1小节的内容可以知道，MessageQueue类的成员变量mPtr保存的是一个C++层的NativeMessageQueue对象的地址值。因此，第3行代码就可以安全地将参数ptr转换成一个NativeMessageQueue对象，接着第4行代码再调用这个NativeMessageQueue对象的成员函数pollOnce来检查当前线程是否有新的消息需要处理。

Step 4：NativeMessageQueue.pollOnce

frameworks/base/core/jni/android_os_MessageQueue.cpp

```
1 void NativeMessageQueue::pollOnce(int timeoutMillis) {
2     mLooper->pollOnce(timeoutMillis);
3 }
```

从前面13.1小节的内容可以知道，NativeMessageQueue类的成员变量mLooper指向了一个C++层的Looper对象，第2行代码调用它的成员函数pollOnce来检查当前线程是否有新的消息需要处理。

Step 5：Looper.pollOnce

frameworks/base/libs/utils/Looper.cpp

```
01 int Looper::pollOnce(int timeoutMillis, int* outFd, int* outEvents, void** outData) {
02     int result = 0;
```

```
03      for (;;) {
04          ......
05
06          if (result != 0) {
07              ......
08
09              return result;
10          }
11
12          result = pollInner(timeoutMillis);
13      }
14  }
```

第3行到第13行的for循环不断地调用成员函数pollInner来检查当前线程是否有新的消息需要处理。如果有新的消息需要处理，那么成员函数pollInner的返回值就不会等于0，这时候第9行代码就会跳出第3行到第13行的for循环，以便当前线程可以对新的消息进行处理。

Step 6：Looper.pollInner

frameworks/base/libs/utils/Looper.cpp

```
01 int Looper::pollInner(int timeoutMillis) {
02      ......
03
04      int result = ALOOPER_POLL_WAKE;
05      ......
06
07      struct epoll_event eventItems[EPOLL_MAX_EVENTS];
08      int eventCount = epoll_wait(mEpollFd, eventItems, EPOLL_MAX_EVENTS, timeoutMillis);
09      ......
10
11      for (int i = 0; i < eventCount; i++) {
12          int fd = eventItems[i].data.fd;
13          uint32_t epollEvents = eventItems[i].events;
14          if (fd == mWakeReadPipeFd) {
15              if (epollEvents & EPOLLIN) {
16                  awoken();
17              }
18              ......
19          }
20          ......
21      }
22      ......
23
24      return result;
25  }
```

在前面的13.1小节中，我们为当前线程在C++层中创建了一个epoll实例，并且将它的文件描述符保存在C++层的Looper类的成员变量mEpollFd中，同时还将一个管道的读端文件描述符注册到它里面，以便可以监听这个管道的IO写事件。

第8行代码调用函数epoll_wait来监听注册在前面所创建的epoll实例中的文件描述符的IO读写事件。如果这些文件描述符都没有发生IO读写事件，那么当前线程就会在函数epoll_wait中进入睡眠等待状态，等待的时间由最后一个参数timeoutMillis来指定。

从函数epoll_wait返回来之后，接下来第11行到第21行的for循环就检查是哪一个文件描述符发生了IO读写事件。第14行的if语句检查发生IO读写事件的文件描述符是否是与当前线程所关联的一个管道的读端文件描述符mWakeReadPipeFd。如果是，并且它所发生的IO读写事件的类型为EPOLLIN，即第15行的if语句为ture，那么这时候就说明其他线程向与当前线程所关联的一个管道写入了一个新的数据。

其他线程在什么情况下会向与当前线程所关联的一个管道写入一个新的数据呢？在接下来的13.3小节中，我们就会看到，当其他线程向当前线程的消息队列发送了一个消息之后，它们就会向与当前线程所关联的一个管道写入一个新的数据，目的是将当前线程唤醒，以便它可以及时地去处理刚刚发送到它的消息队列的消息。

第16行代码调用成员函数awoken将与当前线程所关联的一个管道的数据读出来，以便当前线程下一次可以重新在这个管道上等待其他线程向它的消息队列发送一个新的消息。

Step 7：Looper.awoken

frameworks/base/libs/utils/Looper.cpp

```
1 void Looper::awoken() {
2     ......
3
4     char buffer[16];
5     ssize_t nRead;
6     do {
7         nRead = read(mWakeReadPipeFd, buffer, sizeof(buffer));
8     } while ((nRead == -1 && errno == EINTR) || nRead == sizeof(buffer));
9 }
```

第7行代码调用函数read将与当前线程所关联的一个管道的数据读出来。从这里就可以看出，当前线程根本不关心写入到与它所关联的管道的数据是什么，它只是简单地将这些数据读取出来，以便可以清理这个管道中的旧数据。这样当前线程在下一次消息循环时，如果没有新的消息需要处理，那么它就可以通过监听这个管道的IO写事件进入睡眠等待状态，直到其他线程向它的消息队列发送了一个新的消息为止。

假设当前线程没有新的消息需要处理，那么它就会在前面Step 6中通过调用函数epoll_wait进入睡眠等待状态，直到其他线程向它的消息队列发送了一个新的消息为止。接下来，我们就继续分析其他线程向当前线程的消息队列发送消息的过程。

13.3　线程消息发送过程

Android系统提供了一个Handler类，用来向一个线程的消息队列发送一个消息，它的实现如图13-3如示。

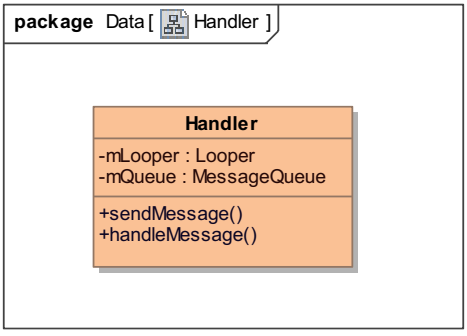


图13-3　Handler类的实现

Handler类内部有mLooper和mQueue两个成员变量，它们分别指向一个Looper对象和一个MessageQueue对象。Handler类还有sendMessage和handleMessage两个成员函数，其中，成员函数sendMessage用来向成员变量mQueue所描述的一个消息队列发送一个消息；而成员函数handleMessage用来处理这个消息，并且它是在与成员变量mLooper所关联的线程中被调用的。

我们一般使用Handler类的默认构造函数来创建一个Handler对象，这时候它的成员变量mLooper和mQueue就会分别指向与当前线程所关联的一个Looper对象和一个MessageQueue对象，如下所示。

frameworks/base/core/java/android/os/Handler.java

```
01 public class Handler {
02     ......
03
04     public Handler() {
05         ......
06
07         mLooper = Looper.myLooper();
08         ......
09
10         mQueue = mLooper.mQueue;
11         ......
12     }
13
14     final MessageQueue mQueue;
15     final Looper mLooper;
16     ......
17 }
```

接下来，我们就从Handler类的成员函数sendMessage开始，分析向一个线程的消息队列发送一个消息的过程，如图13-4所示。

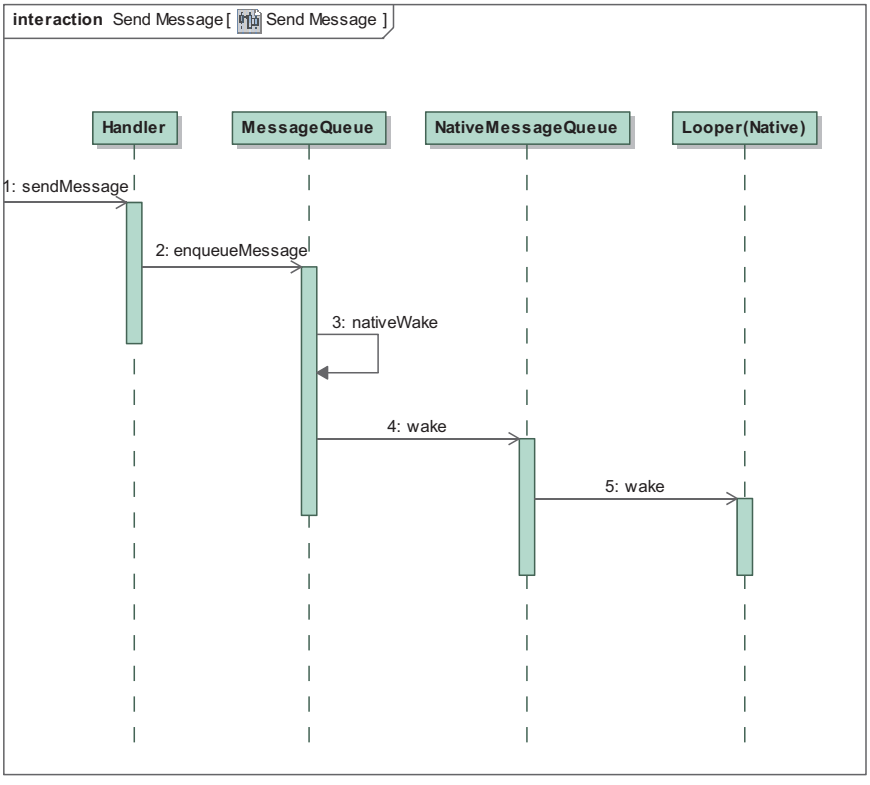


图13-4　线程消息发送过程

这个过程一共分为5个步骤，下面我们就详细分析每一个步骤。

Step 1：Handler.sendMessage

frameworks/base/core/java/android/os/Handler.java

```
01 public class Handler {
02     ......
03
```

```
04      public final boolean sendMessage(Message msg)
05      {
06          return sendMessageDelayed(msg, 0);
07      }
08
09      public final boolean sendMessageDelayed(Message msg, long delayMillis)
10      {
11          if (delayMillis < 0) {
12              delayMillis = 0;
13          }
14          return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);
15      }
16
17      public boolean sendMessageAtTime(Message msg, long uptimeMillis)
18      {
19          boolean sent = false;
20          MessageQueue queue = mQueue;
21          if (queue != null) {
22              msg.target = this;
23              sent = queue.enqueueMessage(msg, uptimeMillis);
24          }
25          ......
26
27          return sent;
28      }
29
30      ......
31  }
```

除了成员函数sendMessage之外，Handler类还有另外两个成员函数sendMessageDelayed和sendMessageAtTime可以用来向一个线程的消息队列发送一个消息。成员函数sendMessage发送的消息的处理时间为系统的当前时间，而成员函数sendMessageDelayed和sendMessageAtTime发送的消息的处理时间是将来的一个时间。其中，前者指定的消息处理时间是一个相对现在的时间，而后者指定的消息处理时间是将来的一个绝对时间。

在成员函数sendMessageAtTime中，参数msg用来描述一个即将发送的消息，而参数uptimeMillis则用来描述这个即将发送的消息的处理时间。第22行代码首先将参数msg的成员变量target设置为当前正在处理的一个Handler对象，接着第23行代码再将参数msg所描述的一个消息发送到一个消息队列中，这个消息队列是由当前正在处理的Handler对象的成员变量mQueue来描述的。

Step 2：MessageQueue. enqueueMessage

frameworks/base/core/java/android/os/MessageQueue.java

```
01 public class MessageQueue {
02      ......
03
04      final boolean enqueueMessage(Message msg, long when) {
05          ......
06
07          final boolean needWake;
08          synchronized (this) {
09              ......
10
11              msg.when = when;
12              //Log.d("MessageQueue", "Enqueing: " + msg);
13              Message p = mMessages;
14              if (p == null || when == 0 || when < p.when) {
15                  msg.next = p;
16                  mMessages = msg;
17                  needWake = mBlocked; // new head, might need to wake up
```

```
18              } else {
19                  Message prev = null;
20                  while (p != null && p.when <= when) {
21                      prev = p;
22                      p = p.next;
23                  }
24                  msg.next = prev.next;
25                  prev.next = msg;
26                  needWake = false; // still waiting on head, no need to wake up
27              }
28
29          }
30          if (needWake) {
31              nativeWake(mPtr);
32          }
33          return true;
34      }
35
36      ......
37  }
```

由于一个消息队列中的消息是按照它们的处理时间从小到大的顺序来排列的，因此，当我们将一个消息发送到一个消息队列时，需要先根据这个消息的处理时间找到它在目标消息队列的合适位置，然后再将它插入到目标消息队列中。

我们分四种情况来讨论如何将一个消息插入到一个目标消息队列中。

（1）目标消息队列是一个空队列。

（2）插入的消息的处理时间等于0。

（3）插入的消息的处理时间小于保存在目标消息队列头的消息的处理时间。

（4）插入的消息的处理时间大于等于保存在目标消息队列头的消息的处理时间。

对于前面三种情况来说，要插入的消息都需要保存在目标消息队列的头部，如第15行到第17行代码所示。

注意

如果一个消息的处理时间等于0，那么就说明这是一个优先级最高的消息，因此我们需要将它保存在目标消息队列的头部，以便它可以优先得到处理。

对于最后一种情况来说，要插入的消息需要保存在目标消息队列中间的某一个位置上，如第19行到第26行代码所示，其中，这个位置是通过第20行到第23行的while循环找到的。

注意

如果要插入的消息的处理时间与目标消息队列中的某一个消息的处理时间相同，那么后来插入的消息会被保存在后面，这样就可以保证先发送的消息可以先获得处理。

一个线程将一个消息插入到一个目标消息队列之后，可能需要将目标线程唤醒，这需要分两种情况来讨论。

（1）插入的消息在目标消息队列中间。

（2）插入的消息在目标消息队列头部。

对于第一种情况来说，由于保存在目标消息队列头部的消息没有发生变化，因此，当前线程无论如何都不需要对目标线程执行唤醒操作，这时候第26行代码就会将变量needWake的值设置为false。

对于第二种情况来说，由于保存在目标消息队列头部的消息发生了变化，因此，当前线程就需要将目标线程唤醒，以便它可以对保存在目标消息队列头部的新消息进行处理。但是，如果这时

候目标线程不是正处于睡眠等待状态，那么当前线程就不需要对它执行唤醒操作。当前正在处理的MessageQueue对象的成员变量mBlocked记录了目标线程是否正处于睡眠等待状态。如果它的值等于true，那么就表示目标线程正处于睡眠等待状态，这时候当前线程就需要将它唤醒。第17行代码将这个成员变量的值保存在变量needWake中，以便当前线程接下来可以决定是否需要将目标线程唤醒。

将要发送的消息插入到目标消息队列中之后，第30行的if语句就检查变量needWake的值是否等于true。如果等于，那么接下来第31行就会调用MessageQueue类的成员函数nativeWake将目标线程唤醒。

假设目标线程正处于睡眠等待状态，接下来我们就继续分析MessageQueue类的成员函数nativeWake的实现。

Step 3：MessageQueue. nativeWake

frameworks/base/core/jni/android_os_MessageQueue.cpp

```
1 static void android_os_MessageQueue_nativeWake(JNIEnv* env, jobject obj, jint ptr) {
2     NativeMessageQueue* nativeMessageQueue = reinterpret_cast<NativeMessageQueue*>(ptr);
3     return nativeMessageQueue->wake();
4 }
```

MessageQueue类的成员函数nativeWake是一个JNI方法，它是由C++层的函数android_os_MessageQueue_nativeWake来实现的。

参数obj指向了一个Java层的MessageQueue对象，而参数ptr指向了这个MessageQueue对象的成员变量mPtr。从前面13.1小节的内容可以知道，MessageQueue类的成员变量mPtr中保存的是一个C++层的NativeMessageQueue对象的地址值。因此，第2行代码就可以安全地将参数ptr转换成一个NativeMessageQueue对象，接着第3行代码再调用这个NativeMessageQueue对象的成员函数wake来唤醒目标线程，以便它可以处理它的消息队列中的新消息。

Step 4：NativeMessageQueue.wake

frameworks/base/core/jni/android_os_MessageQueue.cpp

```
1 void NativeMessageQueue::wake() {
2     mLooper->wake();
3 }
```

从前面13.1小节的内容可以知道，NativeMessageQueue类的成员变量mLooper指向了一个C++层的Looper对象，第2行代码调用它的成员函数wake来唤醒目标线程，以便它可以处理它的消息队列中的新消息。

Step 5：Looper.wake

frameworks/base/libs/utils/Looper.cpp

```
01 void Looper::wake() {
02     ......
03
04     ssize_t nWrite;
05     do {
06         nWrite = write(mWakeWritePipeFd, "W", 1);
07     } while (nWrite == -1 && errno == EINTR);
08
09     .......
10 }
```

从前面13.1小节的内容可以知道，C++层的Looper类的成员变量mWakeWritePipeFd是用来描述一个管道的写端文件描述符的。第6行代码调用函数write向它写入一个“W”字符，即向它所描述的管道写入一个新的数据，这时候目标线程就会因为这个管道发生了一个IO写事件而被唤醒，如前面13.2小节的Step 6所示。

至此，一个线程消息的发送过程就分析完成了。接下来，我们继续分析一个线程消息的处理过程。

13.4 线程消息处理过程

从前面13.2小节的内容可以知道，当一个线程没有新的消息需要处理时，它就会在C++层的Looper类的成员函数pollInner中进入睡眠等待状态，因此，当这个线程有新的消息需要处理时，它首先会在C++层的Looper类的成员函数pollInner中被唤醒，然后沿着之前的调用路径一直返回到Java层的Looper类的静态成员函数loop中，最后就可以对新的消息进行处理了。

接下来，我们就从Java层的Looper类的静态成员函数loop开始，分析一个线程消息的处理过程，如图13-5所示。

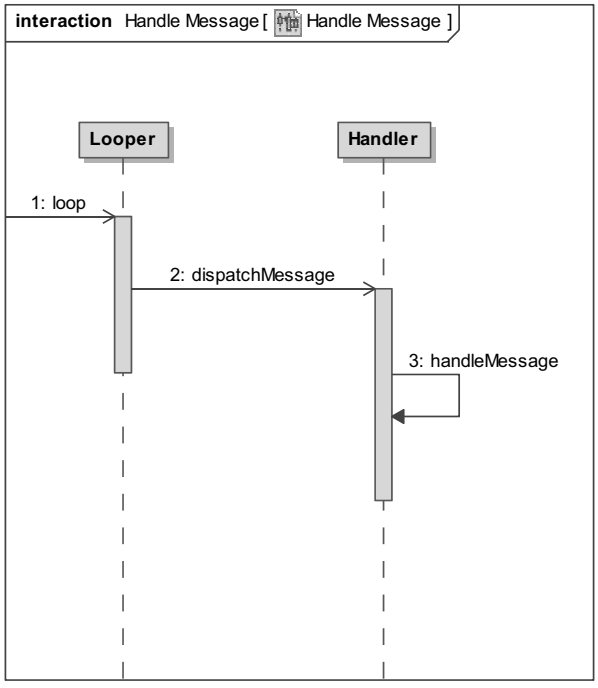


图13-5 线程消息处理过程

这个过程一共分为3个步骤，下面我们就详细分析每一个步骤。

Step 1：Looper.loop

frameworks/base/core/java/android/os/Looper.java

```
01 public class Looper {
02     ......
03
04     public static final void loop() {
05         Looper me = myLooper();
06         MessageQueue queue = me.mQueue;
07         ......
08
09         while (true) {
10             Message msg = queue.next(); // might block
11             ......
12
13             if (msg != null) {
14                 if (msg.target == null) {
15                     // No target is a magic identifier for the quit message.
```

```
16                      return;
17                  }
18                  ......
19
20                  msg.target.dispatchMessage(msg);
21
22                  ......
23              }
24          }
25      }
26
27      ......
28  }
```

第10行代码将要处理的消息保存在Message对象msg中，接着第14行的if语句检查这个Message对象msg的成员变量target是否等于null。如果等于null，那么就说明要处理的是一个退出消息，因此，第16行代码就会退出消息循环；否则，第20行代码就将这个消息分发给Message对象msg的成员变量target来处理。

从前面13.3小节中的Step 1可以知道，Message对象msg的成员变量target指向的是一个Handler对象，因此，接下来我们就继续分析Handler类的成员函数dispatchMessage的实现。

Step 2：Handler. dispatchMessage

frameworks/base/core/java/android/os/Handler.java

```
01 public class Handler {
02     ......
03
04     public void dispatchMessage(Message msg) {
05         if (msg.callback != null) {
06             handleCallback(msg);
07         } else {
08             if (mCallback != null) {
09                 if (mCallback.handleMessage(msg)) {
10                     return;
11                 }
12             }
13             handleMessage(msg);
14         }
15     }
16     ......
17
18     final Callback mCallback;
19     ......
20 }
```

Handler类的成员函数dispatchMessage按照以下顺序来分发一个消息。

（1）如果要处理的消息在发送时指定了一个回调接口，即第5行的if语句为true，那么第6行代码就会调用Handler类的成员函数handleCallback来处理这个消息。

（2）如果条件1不满足，并且负责分发消息的Handler对象的成员变量mCallback指向了一个回调接口，即第8行的if语句为true，那么第9行的if语句就会调用这个回调接口的成员函数handleMessage来处理这个消息。

（3）如果条件2不满足，或者负责分发消息的Handler对象的成员变量mCallback所指向的一个回调接口希望这个消息可以继续向下处理，即第9行的if语句为false，那么第13行代码就会调用Handler类的成员函数handleMessage来处理这个消息。

下面我们就先分析前两种情况的消息处理过程，然后在Step 3中再分析第三种情况的消息处理过程。

Handler类的成员函数handleCallback的实现如下所示。

frameworks/base/core/java/android/os/Handler.java

```
1 public class Handler {
2     ......
3
4     private final void handleCallback(Message message) {
5         message.callback.run();
6     }
7
8     ......
9 }
```

第5行代码调用消息对象message的成员变量callback的成员函数run来处理一个消息。Message类的成员变量callback指向的是一个Runnable对象，因此，第5行代码实际上是将一个消息交给了Runnable类的成员函数run来处理。

当我们调用Handler类的成员函数post来发送一个消息时，Handler类就会为这个消息指定一个回调接口，如下所示。

frameworks/base/core/java/android/os/Handler.java

```
01 public class Handler {
02     ......
03
04     public final boolean post(Runnable r)
05     {
06        return  sendMessageDelayed(getPostMessage(r), 0);
07     }
08
09     ......
10 }
```

参数r是一个类型为Runnable的对象，第6行代码在将它发送到一个消息队列之前，首先会调用Handler类的成员函数getPostMessage将它封装成一个Message对象，这是因为消息队列只接收类型为Message的消息。

Handler类的成员函数getPostMessage的实现如下所示。

frameworks/base/core/java/android/os/Handler.java

```
01 public class Handler {
02     ......
03
04     private final Message getPostMessage(Runnable r) {
05         Message m = Message.obtain();
06         m.callback = r;
07         return m;
08     }
09
10     ......
11 }
```

第5行代码首先创建了一个空的消息对象，接着第6行代码将这个消息对象的成员变量callback设置为参数r所描述的一个Runnable对象。这样我们就可以将一个Runnable对象当作一个消息发送到一个消息队列中去处理，这个消息最终分发给这个Runnable对象的成员函数run来处理。

Handler类在内部定义了一个Callback接口，如下所示。

frameworks/base/core/java/android/os/Handler.java

```
1 public class Handler {
2     ......
3
4     public interface Callback {
5         public boolean handleMessage(Message msg);
6     }
7
8     ......
9 }
```

这个接口只有一个成员函数handleMessage，它是用来处理一个消息的。

除了可以使用Handler类的默认构造函数来创建一个Handler对象之外，我们还可以使用Handler类的另外一个构造函数来创建一个Handler对象，这个构造函数需要指定一个实现了Callback接口的对象，如下所示。

frameworks/base/core/java/android/os/Handler.java

```
01 public class Handler {
02     ......
03
04     public Handler(Callback callback) {
05         ......
06
07         mLooper = Looper.myLooper();
08         ......
09         mQueue = mLooper.mQueue;
10         mCallback = callback;
11     }
12
13     ......
14 }
```

第7行和第9行代码除了会将与当前线程所关联的一个Looper对象和一个MessageQueue对象保存在Handler类的成员变量mLooper和mQueue中之外，第10行代码还会将参数callback所描述的一个Callback对象保存在Handler类的成员变量mCallback中。

这样我们就可以直接使用Handler类来发送消息，并且将这个消息交给一个实现了Callback接口的对象来处理。

以上就是前两种情况的消息处理过程，接下来我们继续分析第三情况的消息处理过程，这是通过调用Handler类的成员函数handleMessage来实现的。

Step 3：Handler. handleMessage

frameworks/base/core/java/android/os/Handler.java

```
01 public class Handler {
02     ......
03
04     /**
05      * Subclasses must implement this to receive messages.
06      */
07     public void handleMessage(Message msg) {
08     }
09
10     ......
11 }
```

Handler类的成员函数handleMessage是一个空实现。一般情况下，我们不直接使用Handler类来发送消息，而是使用它的一个子类来发送消息，这个子类重写了父类Handler的成员函数handleMessage。这样我们就可以创建这个子类的一个对象来发送消息，并且将这个消息交给这个子类对象的成员函数

handleMessage来处理。

至此，一个线程消息的处理过程就分析完成了。这些线程消息都有一个共同的特点，即它们都是先被发送到一个消息队列中，然后再从这个消息队列中取出来处理的。有一种特殊的消息，它们不用事先发送到一个消息队列中，而是由一个线程在空闲的时候主动发送出来。这种特殊的线程消息称为线程空闲消息，接下来我们就分析它的处理过程。

线程空闲消息是由一种称为空闲消息处理器的对象来处理的，这些空闲消息处理器必须要实现一个IdleHandler接口。

IdleHandler接口的定义如下所示。

frameworks/base/core/java/android/os/MessageQueue.java

```
01 public class MessageQueue {
02     ......
03
04     public static interface IdleHandler {
05         /**
06          * Called when the message queue has run out of messages and will now
07          * wait for more.  Return true to keep your idle handler active, false
08          * to have it removed.  This may be called if there are still messages
09          * pending in the queue, but they are all scheduled to be dispatched
10          * after the current time.
11          */
12         boolean queueIdle();
13     }
14
15     ......
16 }
```

它只有一个成员函数queueIdle，用来接收线程空闲消息。

一个空闲消息处理器如果想要接收到一个线程的空闲消息，那么它就必须要注册到这个线程的消息队列中，即注册到与这个线程所关联的一个MessageQueue对象中。

MessageQueue类有一个成员变量mIdleHandlers，它指向了一个IdleHandler列表，用来保存一个线程的空闲消息处理器，如图13-6所示。

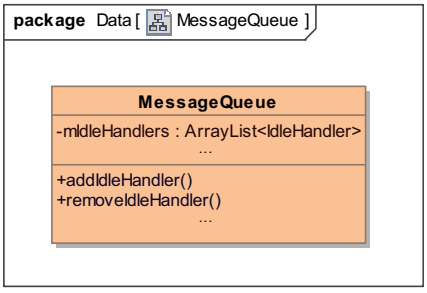


图13-6　空闲消息处理器的注册和注销接口

MessageQueue类有两个成员函数addIdleHandler和removeIdleHandler，分别用来注册和注销一个空闲消息处理器，它们的实现如下所示。

frameworks/base/core/java/android/os/MessageQueue.java

```
01 public class MessageQueue {
02     ......
03
04     public final void addIdleHandler(IdleHandler handler) {
05         if (handler == null) {
06             throw new NullPointerException("Can't add a null IdleHandler");
07         }
08         synchronized (this) {
```

```
            mIdleHandlers.add(handler);
        }
    }

    public final void removeIdleHandler(IdleHandler handler) {
        synchronized (this) {
            mIdleHandlers.remove(handler);
        }
    }

    ......
}
```

从前面13.2小节的Step 2可以知道，一个线程在消息循环的过程中，有时候会变得“无事可做”。当一个线程的消息队列为空，或者保存在消息队列头部的消息的处理时间大于系统的当前时间时，就会发生这种情况。这时候线程就处于一种空闲状态，接下来它就会进入睡眠等待状态。不过，在进入睡眠等待状态之前，这个线程会发出一个线程空闲消息给那些注册了的空闲消息处理器来处理。

下面我们就接着前面13.2小节的Step 2，继续分析MessageQueue类的成员函数next的实现，以便可以了解线程空闲消息的处理过程。

在MessageQueue类的成员函数next中，与线程空闲消息相关的代码如下所示。

frameworks/base/core/java/android/os/MessageQueue.java

```
public class MessageQueue {
    ......

    final Message next() {
        int pendingIdleHandlerCount = -1; // -1 only during first iteration
        ......

        for (;;) {
            ......
            nativePollOnce(mPtr, nextPollTimeoutMillis);

            synchronized (this) {
                ......

                // If first time, then get the number of idlers to run.
                if (pendingIdleHandlerCount < 0) {
                    pendingIdleHandlerCount = mIdleHandlers.size();
                }
                ......

                if (pendingIdleHandlerCount == 0) {
                    ......
                    continue;
                }

                if (mPendingIdleHandlers == null) {
                    mPendingIdleHandlers = new IdleHandler[Math.max(pendingIdleHandlerCount, 4)];
                }
                mPendingIdleHandlers = mIdleHandlers.toArray(mPendingIdleHandlers);
            }

            // Run the idle handlers.
            // We only ever reach this code block during the first iteration.
            for (int i = 0; i < pendingIdleHandlerCount; i++) {
                final IdleHandler idler = mPendingIdleHandlers[i];
                mPendingIdleHandlers[i] = null; // release the reference to the handler

```

```
38                  boolean keep = false;
39                  try {
40                      keep = idler.queueIdle();
41                  } catch (Throwable t) {
42                      Log.wtf("MessageQueue", "IdleHandler threw exception", t);
43                  }
44
45                  if (!keep) {
46                      synchronized (this) {
47                          mIdleHandlers.remove(idler);
48                      }
49                  }
50              }
51
52              // Reset the idle handler count to 0 so we do not run them again.
53              pendingIdleHandlerCount = 0;
54
55              ......
56          }
57      }
58
59      ......
60  }
```

当前线程执行到第16行的if语句时，就说明当前线程准备要进入睡眠等待状态了。

第16行的if语句首先检查变量pendingIdleHandlerCount的值是否小于0。如果小于0，那么第17行代码就会获得所有注册到当前线程的消息队列中的空闲消息处理器的个数，并且保存在变量pendingIdleHandlerCount中。接着第21行的if语句再检查变量pendingIdleHandlerCount的值是否等于0。只有在两种情况下，变量pendingIdleHandlerCount的值才会等于0，其中，第一种情况是没有空闲消息处理器注册到当前线程的消息队列中；第二种情况是之前已经发送过一个线程空闲消息了。如果变量pendingIdleHandlerCount的值等于0，那么当前线程就不会再发出一个线程空闲消息了。

从这里就可以看出，在MessageQueue类的成员函数next的一次调用中，一个线程最多只会发出一个线程空闲消息。

注意

这并不意味着一个线程最多只会发出一个线程空闲消息，因为MessageQueue类的成员函数next在一个线程的消息循环过程中，会不断地被调用，而每一次调用都有可能发出一个线程空闲消息。

第26行到第29行代码首先将注册到当前线程的消息队列中的空闲消息处理器复制到一个空闲消息处理器数组mPendingIdleHandlers中，接着第34行到第50行的for循环再依次调用这个数组中的每一个空闲消息处理器的成员函数queueIdle来接收一个线程空闲消息。

注意

如果一个空闲消息处理器的成员函数queueIdle的返回值等于false，即第45行的if语句为true，那么第47行代码就会将它从MessageQueue类的成员变量mIdleHandlers中移除，这样它以后就不会再接收到线程空闲消息了。

最后，第53行代码将变量pendingIdleHandlerCount的值重新设置为0，这样就可以保证在MessageQueue类的成员函数next的一次调用中，一个线程最多只会发出一个线程空闲消息。

至此，一个线程空闲消息的处理过程就分析完成了。通过注册空闲消息处理器，我们就可以把一些不重要的或者不紧急的事情放在线程空闲的时候来执行，这样就可以充分地利用线程的空闲时间。

第 14 章

Android应用程序的键盘消息处理机制

在Android系统中，键盘事件是由系统来统一监控的，然后再以消息的形式分发给当前激活的应用程序窗口处理。键盘消息与普通消息的区别就在于前者是由硬件中断触发的，而后者则是由应用程序自己产生的。

Android系统的键盘消息是统一由Window管理服务WindowManagerService来管理的。Window管理服务WindowManagerService在启动时，会在内部创建一个输入管理器InputManager，用来监控系统的键盘事件。一个应用程序窗口如果需要接收键盘消息，那么它就必须在激活时与输入管理器InputManager建立一个连接，以便输入管理器InputManager可以将键盘消息分发给它处理。如果一个应用程序窗口在激活时与输入管理器InputManager建立了一个连接，那么当它销毁时，就必须要断开这个连接。应用程序窗口与输入管理器InputManager之间的连接是通过一对输入通道（InputChannel）来描述的，相应地，建立连接和断开连接的过程又分别称为注册InputChannel和注销InputChannel的过程。

接下来，我们将Android应用程序的键盘消息处理机制划分为五个部分的内容来描述，其中，14.1小节介绍Android应用程序的键盘消息处理模型，14.2小节分析输入管理器InputManager的启动过程，14.3小节分析InputChannel的注册过程，14.4小节分析键盘消息的分发过程，14.5小节分析InputChannel的注销过程。

14.1 键盘消息处理模型

我们从输入管理器InputManager开始分析Android应用程序的键盘消息处理模型，它的实现如图14-1所示。

Window管理服务WindowManagerService内部有一个成员变量mInputManager，它指向了Android系统的输入管理器InputManager，这是一个位于Java层中的对象。在C++层中，有一个NativeInputManager对象，它的成员变量mInputManager指向了C++层中的一个输入管理器InputManager。Java层中的输入管理器InputManager就是通过C++层中的输入管理器InputManager来监控系统的键盘事件的。

C++层中的InputManager有两个成员变量mInputDispatcher和mInputReader，它们分别指向了一个InputDispatcher对象和一个InputReader对象，其中，前者用来分发键盘事件给系统当前激活的应用程序窗口，而后者用来监控系统的键盘事件，它们分别运行在一个独立的线程中。

InputDispatcher有两个成员变量mLooper和mFocusedWindow，它们分别指向了一个Looper对象和一个InputWindow对象，其中，前者用来和InputReader通信，而后者用来描述系统当前激活的应用程序窗口。

当一个应用程序窗口被激活时，Window管理服务WindowManagerService就会调用Java层中的InputManager的成员函数setInputWindows将它设置到InputDispatcher的成员变量mFocusedWindow中，以便InputDispatcher

可以将键盘事件分发给它处理。

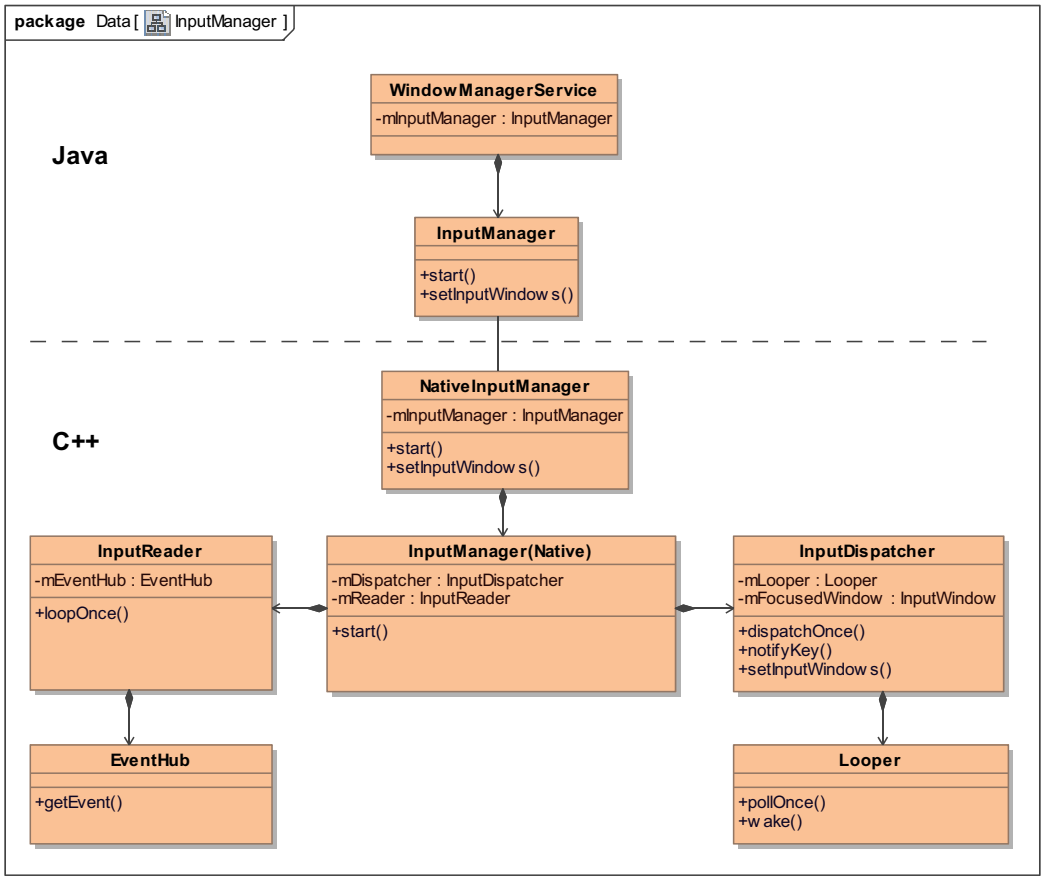


图14-1　InputManager的实现

Java层中的InputManager的成员函数setInputWindows首先会将系统当前激活的应用程序窗口传递给NativeInputManager的成员函数setInputWindows，后者接着再将这个应用程序窗口传递给InputDispatcher的成员函数setInputWindows，这样就可以将系统当前激活的应用程序窗口保存在InputDispatcher的成员变量mFocusedWindow中了。

InputReader有一个成员变量mEventHub，它指向了一个EventHub对象，这个EventHub对象是真正用来监控系统的键盘事件的。

Window管理服务WindowManagerService创建完成Java层中的InputManager之后，接着就会调用它的成员函数start来启动它。Java层中的InputManager在启动的过程中，又会调用NativeInputManager的成员函数start，以便后者可以将C++层中的InputManager启动起来，这是通过调用它的成员函数start来实现的。

C++层中的InputManager在启动的过程中，又会在内部创建两个线程，分别用来启动InputDispatcher和InputReader。

InputDispatcher在启动完成之后，会不断地调用它的成员函数dispatchOnce来检查InputReader是否给它分发了一个键盘事件。如果没有，那么InputDispatcher就会通过调用它的成员变量mLooper的成员函数pollOnce进入睡眠等待状态，直到它被InputReader唤醒为止。

InputReader在启动完成之后，会不断地调用它的成员函数loopOnce来监控系统是否有键盘事件发生。InputReader类的成员函数loopOnce实际上是通过调用成员变量mEventHub的成员函数getEvent来监控系统是否有键盘事件发生的。

当系统发生了键盘事件时，InputReader就会从它的成员变量mEventHub的成员函数getEvent中返

回来，接着再调用InputDispatcher的成员函数notifyKey来唤醒InputDispatcher，以便它可以将刚才发生的键盘事件分给系统当前激活的应用程序窗口处理。InputDispatcher类的成员函数notifyKey实际上是通过调用成员变量mLooper的成员函数wake将InputDispatcher唤醒的。

应用程序窗口与输入管理器InputManager之间的连接模型如图14-2所示。

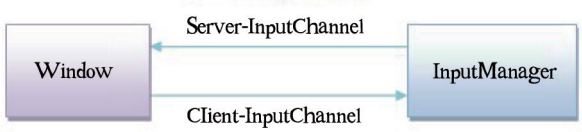


图14-2　应用程序窗口与输入管理器InputManager之间的连接模型

应用程序窗口通过注册两个InputChannel来与输入管理器InputManager建立连接，其中一个InputChannel称为Server端InputChannel，而另外一个InputChannel称为Client端InputChannel。Server端InputChannel需要注册在输入管理器InputManager中，而Client端InputChannel需要注册在应用程序主线程中。

当系统有键盘事件发生时，输入管理器InputManager首先找到系统当前激活的应用程序窗口之前注册到它里面的一个Server端InputChannel，接着再通过这个Server端InputChannel将刚才发生的键盘事件发送给系统当前激活的应用程序窗口处理。

系统当前激活的应用程序窗口在处理完成一个键盘事件之后，就会通过前面注册在应用程序主线程中的一个Client端InputChannel向输入管理器InputManager发送一个键盘事件处理完成通知，以便输入管理器InputManager可以继续将接下来发生的其他键盘事件分发给系统当前激活的应用程序窗口处理。

用来连接输入管理器InputManager和系统当前激活的应用程序窗口的InputChannel的实现如图14-3所示。

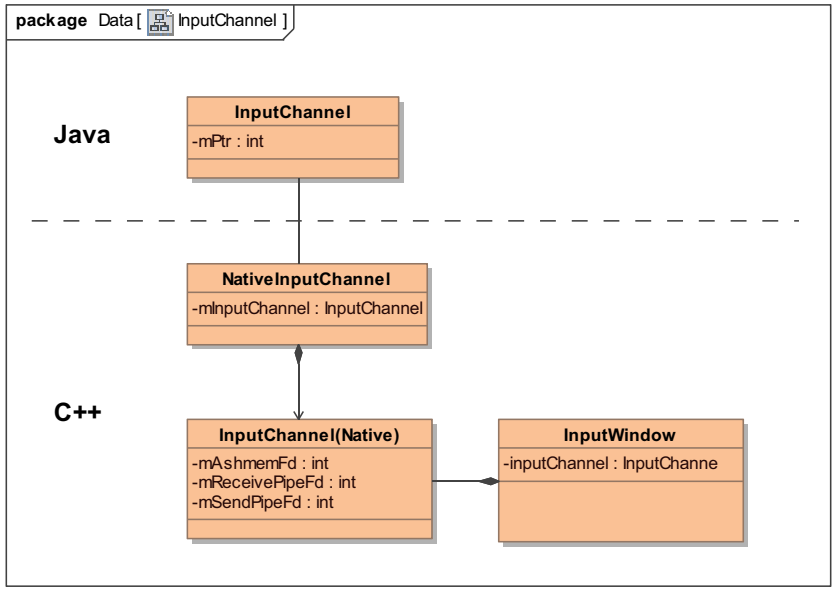


图14-3　InputChannel的实现

Java层中的InputChannel对象在创建时，会同时在C++层中创建一个NativeInputChannel对象，并且将这个NativeInputChannel对象的地址值保存在它的成员变量mPtr中。C++层中的NativeInputChannel对象有一个成员变量mInputChannel，它指向了一个C++层中的InputChannel对象。

C++层中的InputChannel对象包含有三个文件描述符mAshmemFd、mReceivPipeFd和mSendPipeFd。文件描述符mAshmemFd用来描述一块匿名共享内存，这块匿名共享内存是用来在应用程序窗口与输入

管理器InputManager之间传递数据的；文件描述符mReceivPipeFd和mSendPipeFd分别用来描述两个不同的管道的读端和写端，它们是用来在应用程序窗口与输入管理器InputManager之间建立交叉连接的。

C++层中的InputWindow对象是用来描述一个应用程序窗口的，它有一个成员变量inputChannel，指向了C++层中的一个InputChannel对象，这样我们就可以将一个应用程序窗口与一个InputChannel关联起来。

前面在介绍InputDispatcher时提到，它有一个类型为InputWindow的成员变量mFocusedWindow，指向了系统当前激活的应用程序窗口。当系统发生键盘事件时，InputDispatcher就会通过其成员变量mFocusedWindow得到与系统当前激活的应用程序窗口所关联的一个InputChannel，然后再通过这个InputChannel将刚才发生的键盘事件分发给系统当前激活的应用程序窗口处理。

以上就是Android应用程序的键盘消息处理机制的抽象模型。接下来我们再通过InputManager的启动过程、InputChannel的注册过程、键盘消息的分发过程，以及InputChannel的注销过程来进一步理解这个模型。

14.2 InputManager的启动过程

Window管理服务WindowManagerService在启动时，会在内部创建一个InputManager，以便可以统一地管理系统的键盘事件。InputManager在创建的过程中，又会在内部创建一个InputDispatcher和一个InputReader，分别用来分发键盘事件给系统当前激活的应用程序窗口处理，以及监控系统键盘事件的发生。下面我们首先分析InputManager的创建和启动过程，然后分析InputDispatcher和InputReader的启动过程。

14.2.1 创建InputManager

从前面11.3小节的内容可以知道，System进程在启动时，会创建一个类型为WMThread的线程，用来启动Window管理服务WindowManagerService。由于InputManager是在Window管理服务WindowManagerService启动时创建的，因此，接下来我们就从WMThread类的成员函数run开始分析InputManager的创建过程，如图14-4所示。

这个过程一共分为4个步骤，下面我们就详细分析每一个步骤。

Step 1：WMThread.run

frameworks/base/services/java/com/android/server/WindowManagerService.java

```
01 public class WindowManagerService extends IWindowManager.Stub
02         implements Watchdog.Monitor {
03     ......
04
05     static class WMThread extends Thread {
06         ......
07
08         public void run() {
09             ......
10
11             WindowManagerService s = new WindowManagerService(mContext, mPM,
12                 mHaveInputMethods);
13             ......
14         }
15     }
16
17     ......
18 }
```

第11行和第12行代码在WMThread线程中创建了一个WindowManagerService实例，用来作为系统的Window管理服务。

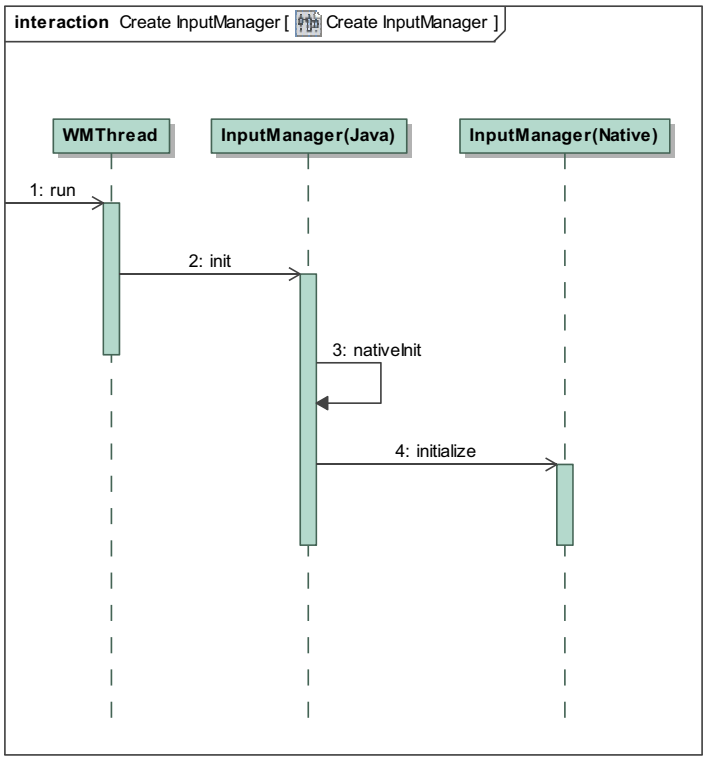


图14-4　InputManager的创建过程

Window管理服务WindowManagerService的创建过程如下所示。

frameworks/base/services/java/com/android/server/WindowManagerService.java

```
01 public class WindowManagerService extends IWindowManager.Stub
02         implements Watchdog.Monitor {
03     ......
04
05     final InputManager mInputManager;
06     ......
07
08     private WindowManagerService(Context context, PowerManagerService pm,
09             boolean haveInputMethods) {
10         ......
11
12         mInputManager = new InputManager(context, this);
13         ......
14
15         mInputManager.start();
16
17         ......
18     }
19
20     ......
21 }
```

第12行代码首先创建了一个InputManager实例，用来作为系统的输入管理器，并且保存在Window管理服务WindowManagerService的成员变量mInputManager中；接着第15行代码调用它的成员函数start将它启动起来，以便可以管理系统的键盘事件。

下面我们主要分析输入管理器InputManager的创建过程，在接下来的14.2.2小节中，再分析它的启动过程。

输入管理器InputManager的创建过程如下所示。

frameworks/base/services/java/com/android/server/InputManager.java

```
01 public class InputManager {
02     ......
03
04     public InputManager(Context context, WindowManagerService windowManagerService) {
05         ......
06
07         init();
08     }
09
10     ......
11 }
```

第7行代码调用了InputManager类的成员函数init来初始化前面所创建的输入管理器InputManager。

Step 2：InputManager.init

frameworks/base/services/java/com/android/server/InputManager.java

```
01 public class InputManager {
02     ......
03
04     private static native void nativeInit();
05     ......
06
07     private void init() {
07         Slog.i(TAG, "Initializing input manager");
09         nativeInit(mCallbacks);
10     }
11
12     ......
13 }
```

第9行代码调用了InputManager类的成员函数nativeInit，以便可以在C++层中创建一个NativeInputManager对象。

Step 3：InputManager.nativeInit

frameworks/base/services/jni/com_android_server_InputManager.cpp

```
01 static sp<NativeInputManager> gNativeInputManager;
02 ......
03
04 static void android_server_InputManager_nativeInit(JNIEnv* env, jclass clazz,
05         jobject callbacks) {
06     if (gNativeInputManager == NULL) {
07         gNativeInputManager = new NativeInputManager(callbacks);
08     }
09     ......
10 }
```

InputManager类的成员函数nativeInit是一个JNI方法，它是由C++层中的函数android_server_InputManager_nativeInit来实现的。

第6行的if语句首先检查全局变量gNativeInputManager的值是否等于null。如果等于null，那么第7行代码就会创建一个NativeInputManager对象，并且将它保存在全局变量gNativeInputManager中。

NativeInputManager对象的创建过程如下所示。

frameworks/base/services/jni/com_android_server_InputManager.cpp

```
1 NativeInputManager::NativeInputManager(jobject callbacksObj) :
2     mFilterTouchEvents(-1), mFilterJumpyTouchEvents(-1), mVirtualKeyQuietTime(-1),
3     mMaxEventsPerSecond(-1),
4     mDisplayWidth(-1), mDisplayHeight(-1), mDisplayOrientation(ROTATION_0) {
5     ......
6
7     sp<EventHub> eventHub = new EventHub();
8     mInputManager = new InputManager(eventHub, this, this);
9 }
```

第7行代码首先创建了一个EventHub对象，接着第8行代码再以这个EventHub对象为参数来创建一个C++层中的InputManager对象，并且保存在NativeInputManager类的成员变量mInputManager中。

C++层中的InputManager对象的创建过程如下所示。

frameworks/base/libs/ui/InputManager.cpp

```
1 InputManager::InputManager(
2         const sp<EventHubInterface>& eventHub,
3         const sp<InputReaderPolicyInterface>& readerPolicy,
4         const sp<InputDispatcherPolicyInterface>& dispatcherPolicy) {
5     mDispatcher = new InputDispatcher(dispatcherPolicy);
6     mReader = new InputReader(eventHub, readerPolicy, mDispatcher);
7     initialize();
8 }
```

第5行和第6行代码首先创建了一个InputDispatcher对象和一个InputReader对象，并且将它们保存在InputManager类的成员变量mDispatcher和mReader中；接着第7行代码调用InputManager类的成员函数initialize来初始化前面所创建的一个InputManager对象。

Step 4：InputManager.initialize

frameworks/base/libs/ui/InputManager.cpp

```
1 void InputManager::initialize() {
2     mReaderThread = new InputReaderThread(mReader);
3     mDispatcherThread = new InputDispatcherThread(mDispatcher);
4 }
```

第2行和第3行代码分别以InputManager类的成员变量mReader和mDispatcher为参数来创建一个InputReaderThread线程和一个InputDispatcherThread线程。这两个线程分别是用来运行前面所创建的一个InputReader对象和一个InputDispatcher对象的。

至此，Java层中的InputManager，以及C++层中的InputManager的创建过程就分析完成了。接下来我们再分析它们的启动过程。

14.2.2　启动InputManager

从前面14.2.1小节的Step 1可以知道，Window管理服务WindowManagerService创建完成InputManager之后，就会调用它的成员函数start，以便可以将它启动起来，如图14-5所示。

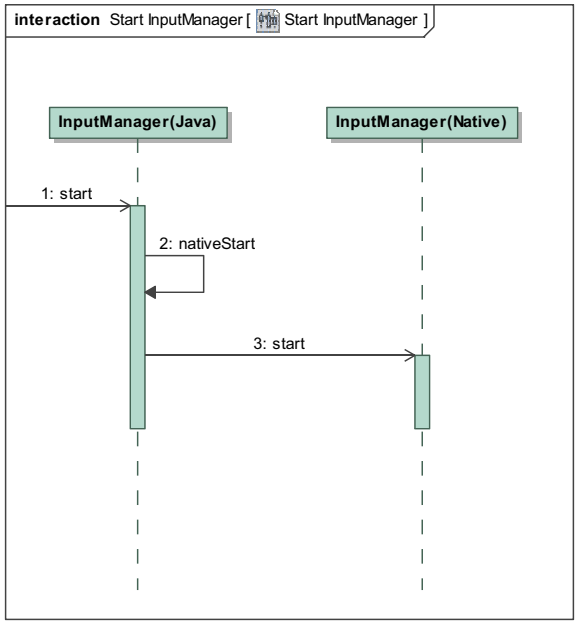


图14-5　InputManager的启动过程

这个过程一共分为3个步骤，下面我们就详细分析每一个步骤。

Step 1：InputManager.start

frameworks/base/services/java/com/android/server/InputManager.java

```
01 public class InputManager {
02     ......
03
04     private static native void nativeStart();
05     ......
06
07     public void start() {
08         Slog.i(TAG, "Starting input manager");
09         nativeStart();
10     }
11
12     ......
13 }
```

第9行代码调用Java层中的InputManager类的成员函数nativeStart将C++层中的InputManager启动起来，以便它可以管理系统的键盘事件。

Step 2：InputManager.nativeStart

frameworks/base/services/jni/com_android_server_InputManager.cpp

```
1 static void android_server_InputManager_nativeStart(JNIEnv* env, jclass clazz) {
2     ......
3
4     status_t result = gNativeInputManager->getInputManager()->start();
5
6     ......
7 }
```

Java层中的InputManager类的成员函数nativeStart是一个JNI方法，它是由C++层中的函数android_server_InputManager_nativeStart来实现的。

从前面14.2.1小节的Step 3可以知道，全局变量gNativeInputManager指向了一个NativeInputManager

对象。第4行代码首先调用它的成员函数getInputManager来获得C++层中的InputManager对象，接着再调用这个InputManager对象的成员函数start将C++层中的InputManager启动起来。

Step 3：InputManager.start

frameworks/base/libs/ui/InputManager.cpp

```
1 status_t InputManager::start() {
2     status_t result = mDispatcherThread->run("InputDispatcher", PRIORITY_URGENT_DISPLAY);
3     ......
4
5     result = mReaderThread->run("InputReader", PRIORITY_URGENT_DISPLAY);
6     ......
7
8     return OK;
9 }
```

InputManager类的成员变量mDispatcherThread和mReaderThread的类型分别为InputDispatcherThread和InputReaderThread，它们分别用来描述一个线程。第2行和第5行代码分别将这两个线程启动起来，以便可以用来分发键盘事件给系统当前激活的应用程序窗口处理，以及监控系统键盘事件的发生。

InputDispatcherThread类和InputReaderThread类都是从Thread类继承下来的，当它们的成员函数run被调用时，系统就会创建一个线程，并且以它们的成员函数threadLoop作为线程的入口函数。

至此，Java层中的InputManager，以及C++层中的InputManager的启动过程就分析完成了。接下来我们再分析C++层中的InputDispatcher和InputReader的启动过程。

14.2.3　启动InputDispatcher

从前面14.2.2小节的Step 3可以知道，InputDispatcher是在一个InputDispatcherThread线程中启动的。下面我们就从这个线程的入口函数，即InputDispatcherThread类的成员函数threadLoop开始分析InputDispatcher的启动过程，如图14-6所示。

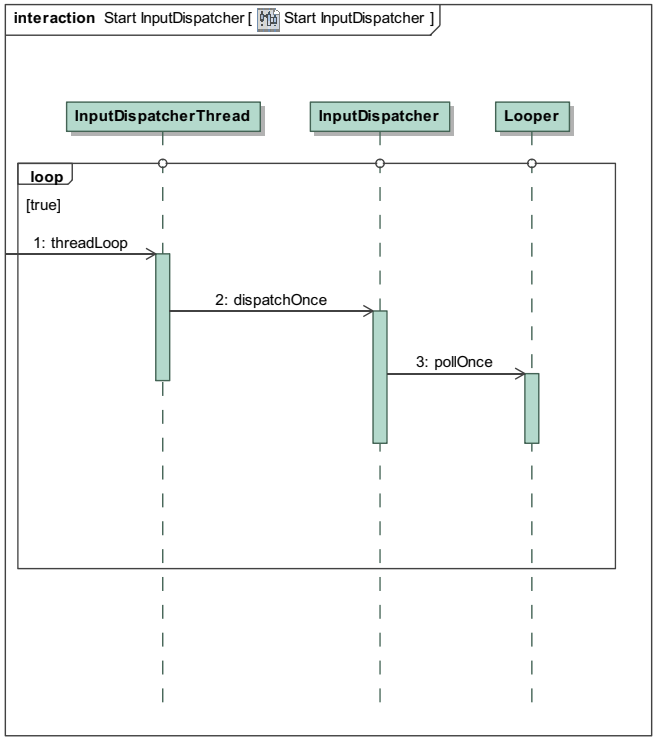


图14-6　InputDispatcher的启动过程

这个过程一共分为3个步骤，下面我们就详细分析每一个步骤。

Step 1：InputDispatcherThread.threadLoop

frameworks/base/libs/ui/InputDispatcher.cpp

```
1 bool InputDispatcherThread::threadLoop() {
2     mDispatcher->dispatchOnce();
3     return true;
4 }
```

InputDispatcherThread类的成员变量mDispatcher指向了在前面14.2.1小节的Step 3中所创建的一个InputDispatcher对象，第2行代码调用它的成员函数dispatchOnce来检查是否有新的键盘事件需要分发给系统当前激活的应用程序窗口处理。

Step 2：InputDispatcher.dispatchOnce

frameworks/base/libs/ui/InputDispatcher.cpp

```
01 void InputDispatcher::dispatchOnce() {
02     nsecs_t keyRepeatTimeout = mPolicy->getKeyRepeatTimeout();
03     nsecs_t keyRepeatDelay = mPolicy->getKeyRepeatDelay();
04
05     nsecs_t nextWakeupTime = LONG_LONG_MAX;
06     { // acquire lock
07         AutoMutex _l(mLock);
08         dispatchOnceInnerLocked(keyRepeatTimeout, keyRepeatDelay, & nextWakeupTime);
09
10         if (runCommandsLockedInterruptible()) {
11             nextWakeupTime = LONG_LONG_MIN;  // force next poll to wake up immediately
12         }
13     } // release lock
14
15     // Wait for callback or timeout or wake.  (make sure we round up, not down)
16     nsecs_t currentTime = now();
17     int32_t timeoutMillis;
18     if (nextWakeupTime > currentTime) {
19         uint64_t timeout = uint64_t(nextWakeupTime - currentTime);
20         timeout = (timeout + 999999LL) / 1000000LL;
21         timeoutMillis = timeout > INT_MAX ? -1 : int32_t(timeout);
22     } else {
23         timeoutMillis = 0;
24     }
25
26     mLooper->pollOnce(timeoutMillis);
27 }
```

InputDispatcher类的成员变量mPolicy指向了在前面14.2.1小节的Step 3中所创建的一个NativeInputManager对象，通过这个对象我们就可以获得一些键盘事件的分发策略。

第2行和第3行代码通过调用成员变量mPolicy的成员函数getKeyRepeatTimeout和getKeyRepeatDelay来获得两个控制键盘重复按键的参数keyRepeatTimeout和keyRepeatDelay。当我们按住一个键盘按键不放时，一般就会产生一个重复的键盘事件。重复的键盘事件有两种产生方式，其中一种方式是由内核中的键盘驱动程序产生的；另外一种方式是由用户空间中的InputDispatcher产生的。如果是由内核中的键盘驱动程序产生的，那么用户空间中的InputDispatcher就什么也不用做；否则，它就会按照以下方式产生一个重复的键盘事件。首先，如果一个键盘按键在keyRepeatTimeout时间内被连续地按住，那么InputDispatcher就认为系统当前发生了一个重复的键盘事件；然后，InputDispatcher就会每隔keyRepeatDelay时间分发一个相同的键盘事件给系统当前激活的应用程序窗口处理，直到前面被按住的键盘按键被松开为止。

注意

参数keyRepeatTimeout和keyRepeatDelay的时间单位都是纳秒。

第8行代码以keyRepeatTimeout和keyRepeatDelay为参数来调用InputDispatcher类的成员函数dispatchOnceInnerLocked，用来检查系统当前是否有新的键盘事件需要分发。如果系统当前没有新的键盘事件需要分发，那么InputDispatcher类的成员函数dispatchOnceInnerLocked就会通过第三个参数nextWakeupTime来通知InputDispatcher，它需要隔一段时间之后再来检查系统是否有新的键盘事件需要分发。在这段时间内，InputDispatcher会进入睡眠等待状态，直到系统当前时间达到nextWakeupTime纳秒为止。在后面的14.4小节中，我们再分析InputDispatcher类的成员函数dispatchOnceInnerLocked的实现。

第10行的if语句调用InputDispatcher类的成员函数runCommandsLockedInterruptible来检查InputDispatcher中是否有一些缓存的命令未执行。由于InputDispatcher接下来就要进入睡眠等待状态了，即它要变得空闲了，因此，这时候就最好用来执行之前所缓存的命令。如果InputDispatcher中有缓存的命令，那么InputDispatcher类的成员函数runCommandsLockedInterruptible在执行了它们之后，就会返回一个true值。在这种情况下，第11行代码会将变量nextWakeupTime的值重新设置为LONG_LONG_MIN，表示InputDispatcher接下来不可以进入睡眠等待状态，因为很有可能在执行缓存命令期间，系统发生了新的键盘事件。

第16行到第24行代码将变量nextWakeupTime所描述的一个绝对时间转换成一个相对时间，并且将它的时间单位从纳秒转换为毫秒，最后就可以得到InputDispatcher要进入睡眠等待状态的时间，保存在变量timeoutMillis中。

InputDispatcher类的成员变量mLooper指向了C++层中的一个Looper对象，第26行代码调用它的成员函数pollOnce使得InputDispatcher进入到睡眠等待状态，并且将它的睡眠等待状态的最长时间设置为timeoutMillis毫秒。

Step 3：Looper.pollOnce

第3步可以参考前面13.2小节中的第5步。在我们这个场景中，InputDispatcher会在这一步中进入睡眠等待状态，直到达到指定的时间为止。

注意

在InputDispatcher处于睡眠等待状态期间，如果系统发生了键盘事件，那么InputReader就会提前将InputDispatcher唤醒，以便它可以将刚才发生的键盘事件分发给系统当前激活的应用程序窗口处理。

至此，InputDispatcher的启动过程就分析完成了。假设系统当前没有键盘事件发生，那么InputDispatcher在启动起来之后，就会马上进入睡眠等待状态，直到它被InputReader唤醒为止，即系统发生了新的键盘事件为止。在后面的14.4小节中，我们再分析InputReader是如何将InputDispatcher唤醒的。

14.2.4　启动InputReader

从前面14.2.2小节的Step 3可以知道，InputReader是在一个InputReaderThread线程中启动的。接下来我们就从这个线程的入口函数，即InputReaderThread类的成员函数threadLoop开始分析InputReader的启动过程，如图14–7所示。

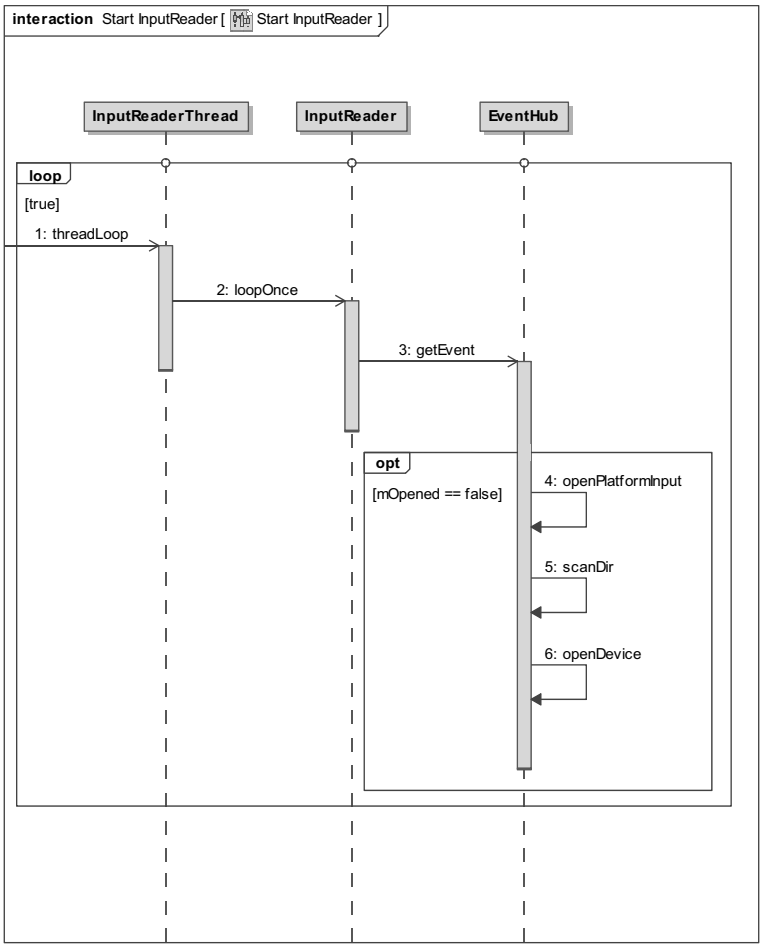


图14–7　InputReader的启动过程

这个过程一共分为6个步骤，下面我们就详细分析每一个步骤。

Step 1：InputReaderThread.threadLoop

frameworks/base/libs/ui/InputReader.cpp

```
1 bool InputReaderThread::threadLoop() {
2     mReader->loopOnce();
3     return true;
4 }
```

InputReaderThread类的成员变量mReader指向了在前面14.2.1小节的Step 3中所创建的一个InputReader对象，第2行代码调用它的成员函数loopOnce来监控系统是否有新的键盘事件发生。

Step 2：InputReader.loopOnce

frameworks/base/libs/ui/InputReader.cpp

```
1 void InputReader::loopOnce() {
2     RawEvent rawEvent;
3     mEventHub->getEvent(& rawEvent);
4     ......
5
6     process(& rawEvent);
7 }
```

InputReader类的成员变量mEventHub指向了在前面14.2.1小节的Step 3中所创建的一个EventHub对

象，第3行代码首先调用它的成员函数getEvent来获得一个键盘事件，并且保存在一个RawEvent结构体rawEvent中；接着第6行代码再调用InputReader类的成员函数process来处理这个键盘事件。在后面的14.4小节中，我们再分析InputReader类的成员函数process的实现。

接下来，我们就分析EventHub类的成员函数getEvent的实现，以便可以了解它是如何监控系统的键盘事件的。

Step 3：EventHub.getEvent

我们分段来阅读EventHub类的成员函数getEvent。

frameworks/base/libs/ui/EventHub.cpp

```
01 bool EventHub::getEvent(RawEvent* outEvent)
02 {
03     outEvent->deviceId = 0;
04     outEvent->type = 0;
05     outEvent->scanCode = 0;
06     outEvent->keyCode = 0;
07     outEvent->flags = 0;
08     outEvent->value = 0;
09     outEvent->when = 0;
```

第3行到第9行代码用来初始化参数outEvent所描述的一个键盘事件。

frameworks/base/libs/ui/EventHub.cpp

```
10     if (!mOpened) {
11         mError = openPlatformInput() ? NO_ERROR : UNKNOWN_ERROR;
12         mOpened = true;
13         ......
14     }
```

EventHub类的成员函数getEvent在监控系统的键盘事件之前，首先要打开系统中的输入设备。EventHub类有一个成员变量mOpened，它是一个布尔值，当EventHub类打开了系统的输入设备之后，它的值就会等于true；否则，它的值就会等于false。

第10行的if语句判断EventHub类是否已经打开了系统的输入设备。如果还没有打开，那么接下来第11行代码就会调用EventHub类的成员函数openPlatformInput来打开它们，并且第12行代码会将EventHub类的成员变量mOpened的值设置为true。在接下来的Step 4中，我们再分析EventHub类的成员函数openPlatformInput的实现。

将系统的输入设备打开之后，EventHub类就会得到一系列的文件描述符，并且将它们保存在成员变量mFDs所描述的一个pollfd数组中，数组的大小保存在成员变量mFDCount中。同时，每一个被打开的输入设备还使用一个device_t结构体来描述，保存在EventHub类的成员变量mDevices所描述的一个device_t结构体数组中。

frameworks/base/libs/ui/EventHub.cpp

```
15     for (;;) {
16         ......
17
18         // Grab the next input event.
19         for (;;) {
20             // Consume buffered input events, if any.
21             if (mInputBufferIndex < mInputBufferCount) {
22                 const struct input_event& iev = mInputBufferData[mInputBufferIndex++];
23                 const device_t* device = mDevices[mInputDeviceIndex];
24
25                 ......
```

```
26                  if (device->id == mFirstKeyboardId) {
27                      outEvent->deviceId = 0;
28                  } else {
29                      outEvent->deviceId = device->id;
30                  }
31                  outEvent->type = iev.type;
32                  outEvent->scanCode = iev.code;
33                  if (iev.type == EV_KEY) {
34                      status_t err = device->layoutMap->map(iev.code,
35                          & outEvent->keyCode, & outEvent->flags);
36                      ......
37                  } else {
38                      outEvent->keyCode = iev.code;
39                  }
40                  outEvent->value = iev.value;
41
42                  ......
43                  outEvent->when = systemTime(SYSTEM_TIME_MONOTONIC);
44                  return true;
45              }
```

第15行到后面第77行的for循环用来监控前面所打开的输入设备是否有IO输入事件发生。对于键盘输入设备来说，要监控的就是是否有键盘按键被按下或者松开了。每当有一个输入设备发生了一个IO输入事件之后，EventHub类就会将这些IO事件数据保存在成员变量mInputBufferData所描述的一个缓冲区中。每一个IO输入事件都使用一个input_event结构体来描述，因此，我们也可以把EventHub类的成员变量mInputBufferData看作是一个input_event结构体数组，它的大小保存在EventHub类的成员变量mInputBufferCount中。

由于EventHub类的成员函数getEvent一次只能处理一个IO输入事件，因此，它就使用成员变量mInputBufferIndex记录下一次要处理的IO输入事件在数组mInputBufferData中的位置。同时，EventHub类还使用成员变量mInputDeviceIndex记录下一次要处理的IO输入事件所对应的输入设备在数组mDevices中的位置。

第19行到后面第66行的for循环用来检查数组mInputBufferData中是否还有未处理的IO输入事件。如果有，即第21行的if语句为true，那么接下来第22行到第44行代码就会将这个IO输入事件的数据保存在输出参数outEvent所描述的一个RawEvent结构体中，以便可以返回给调用者处理。

第22行代码首先获得当前需要处理的IO输入事件，并且保存在input_event结构体iev中；接着第23行代码获得与这个IO输入事件所对应的输入设备，并且保存在device_t结构体 device中。

输出参数outEvent有一个成员变量deviceId，用来描述发生IO输入事件的输入设备的ID值。在EventHub类中，每一个已经打开的输入设备都有一个ID值。一个已经打开的输入设备的ID值是与它的打开顺序相关的，一般来说，越在后面打开的输入设备，它所分配到的ID值就越大。不过，Android系统的内建键盘设备的ID值是恒为0的，与它的打开顺序无关。EventHub类在打开系统的输入设备时，如果发现当前打开的是一个内建的键盘设备，那么就会将分配给它的ID值保存在成员变量mFirstKeyboardId中，以便可以知道它所对应的输入设备是一个内建的键盘设备。第26行的if语句检查当前正在处理的输入设备device的ID值是否等于EventHub类的成员变量mFirstKeyboardId。如果等于，那么接下来第27行代码就需要将输出参数outEvent的成员变量deviceId的值设置为0，以便可以让调用者知道这是一个Android系统内建键盘的输入事件。

结构体input_event有三个成员变量type、code和value，分别用来描述一个IO输入事件的类型、代码和值。例如，对于键盘输入事件来说，成员变量type的值恒为EV_KEY，成员变量code的值等于键

盘扫描码，成员变量value的值等于0或者1，其中，0表示按下，1表示松开。第31行、第32行和第40行代码分别将正在处理的IO输入事件iev的成员变量type、code和value的值设置到输出参数outEvent的成员变量type、scanCode和value中。

注意

对于键盘输入事件来说，我们除了需要知道它的扫描码之后，还需要知道它的键盘码。键盘码是一个与键盘按键布局相关的参数。由于不同的键盘设备可能会有不同的键盘按键布局，因此，对于每一个键盘设备来说，EventHub都使用一个KeyLayoutMap对象来描述它的键盘按键布局，并且保存在用来描述它的一个device_t结构体的成员变量layoutMap中。

第33行的if语句检查正在处理的IO输入事件iev的成员变量type的值是否等于EV_KEY。如果等于，那么就说明正在处理的是一个键盘输入事件。因此，接下来第34行和第35行代码就会调用它所对应的输入设备的成员变量layoutMap的成员函数map将它的扫描码转换成一个键盘码，并且保存在输出参数outEvent的成员变量keyCode中。

最后，第43行代码将输出参数outEvent的成员变量when的值设置为系统的当前时间，用来记录正在处理的IO输入事件的发生时间。

注意

在同一时刻，保存在数组mInputBufferData中的IO输入事件都是发生在同一个输入设备上的。因此，当保存在数组mInputBufferData中的IO输入事件都已经处理完成之后，即前面第21行的if语句为false，函数接下来就需要继续检查其他输入设备是否也有IO输入事件发生。

frameworks/base/libs/ui/EventHub.cpp

```
46              mInputDeviceIndex += 1;
47              if (mInputDeviceIndex >= mFDCount) {
48                  break;
49              }
50
51              const struct pollfd& pfd = mFDs[mInputDeviceIndex];
52              if (pfd.revents & POLLIN) {
53                  int32_t readSize = read(pfd.fd, mInputBufferData,
54                      sizeof(struct input_event) * INPUT_BUFFER_SIZE);
55                  if (readSize < 0) {
56                      if (errno != EAGAIN && errno != EINTR) {
57                          LOGW("could not get event (errno=%d)", errno);
58                      }
59                  } else if ((readSize % sizeof(struct input_event)) != 0) {
60                      LOGE("could not get event (wrong size: %d)", readSize);
61                  } else {
62                      mInputBufferCount = readSize / sizeof(struct input_event);
63                      mInputBufferIndex = 0;
64                  }
65              }
66          }
```

第46行代码首先将EventHub类的成员变量mInputDeviceIndex的值增加1，表示当前正在处理的输入设备在数组mDevices中的位置；接着第47行的if语句检查它的值是否已经大于等于数组mDevices的大小。如果是，那么就说明EventHub类已经检查过系统所有的输入设备是否发生过IO输入事件了。因此，接下来第48行代码就会跳出第19行到第66行的for循环，以便可以重新监控系统的输入设备是否有新的IO输入事件发生；否则，第51行到第65行代码就会将下一个发生了IO输入事件的输入设备的IO输

入事件数据读取出来，并且保存在数组mInputBufferData中，以便接下来可以对它们进行处理。

假设EventHub类已经检查过系统所有的输入设备是否发生过IO输入事件了，即第47行的if语句为true，那么函数接下来就会重新监控系统的输入设备是否有新的IO输入事件发生。

frameworks/base/libs/ui/EventHub.cpp

```
67          mInputDeviceIndex = 0;
68          ......
69
70          release_wake_lock(WAKE_LOCK_ID);
71
72          int pollResult = poll(mFDs, mFDCount, -1);
73
74          acquire_wake_lock(PARTIAL_WAKE_LOCK, WAKE_LOCK_ID);
75
76          ......
77      }
78  }
```

第67行代码首先将EventHub类的成员变量mInputDeviceIndex的值重新设置为0，以便下一次可以从头开始检查系统的输入设备是否有新的IO输入事件发生；接着第72行代码就调用函数poll来监控系统的输入设备是否有IO输入事件发生。

注意

如果这时候系统所有的输入设备都没有IO输入事件发生，那么当前线程就会在函数poll中进入睡眠等待状态，直到任意的一个输入设备发生了IO输入事件为止。

回到前面第11行代码中，接下来我们继续分析EventHub的成员函数openPlatformInput的实现，以便可以了解它是如何打开系统的输入设备的。

Step 4：EventHub.openPlatformInput

frameworks/base/libs/ui/EventHub.cpp

```
01 bool EventHub::openPlatformInput(void)
02 {
03      /*
04       * Open platform-specific input device(s).
05       */
06      int res;
07
08      mFDCount = 1;
09      mFDs = (pollfd *)calloc(1, sizeof(mFDs[0]));
10      mDevices = (device_t **)calloc(1, sizeof(mDevices[0]));
11      mFDs[0].events = POLLIN;
12      mFDs[0].revents = 0;
13      mDevices[0] = NULL;
14  #ifdef HAVE_INOTIFY
15      mFDs[0].fd = inotify_init();
16      res = inotify_add_watch(mFDs[0].fd, device_path, IN_DELETE | IN_CREATE);
17      if(res < 0) {
18          LOGE("could not add watch for %s, %s\n", device_path, strerror(errno));
19      }
20  #else
21      /*
22       * The code in EventHub::getEvent assumes that mFDs[0] is an inotify fd.
23       * We allocate space for it and set it to something invalid.
24       */
25      mFDs[0].fd = -1;
```

```
26 #endif
27
28     res = scanDir(device_path);
29     if(res < 0) {
30         LOGE("scan dir failed for %s\n", device_path);
31     }
32
33     return true;
34 }
```

EventHub类打开的第一个输入设备并不是一个真实的输入设备，而是一个虚拟的输入设备。如果内核支持inotify机制[1]，那么这个虚拟的输入设备就是用来实时监控系统中的输入设备是否发生变化的；否则，这个虚拟的输入设备是不起任何作用的。

如果内核支持inotify机制，那么第15行代码首先会调用函数inotify_init来创建一个inotify实例，接着第16行代码再调用函数inotify_add_watch来监控device_path（/dev/input）目录的变化。当系统新增（IN_CREATE）或者移除（IN_DELETE）了一个输入设备时，目录/dev/input的结构就会发生变化，因此，通过监控这个目录的变化就可以监控系统的输入设备的变化。这样EventHub类就可以实时地在内部打开新增的输入设备，以及关闭已经被移除的输入设备。

最后，第28行代码调用EventHub类的成员函数scanDir来打开device_path（/dev/input）目录下的输入设备文件，以便可以打开系统中的输入设备。

Step 5：EventHub.scanDir

frameworks/base/libs/ui/EventHub.cpp

```
01 int EventHub::scanDir(const char *dirname)
02 {
03     char devname[PATH_MAX];
04     char *filename;
05     DIR *dir;
06     struct dirent *de;
07     dir = opendir(dirname);
08     if(dir == NULL)
09         return -1;
10     strcpy(devname, dirname);
11     filename = devname + strlen(devname);
12     *filename++ = '/';
13     while((de = readdir(dir))) {
14         if(de->d_name[0] == '.' &&
15             (de->d_name[1] == '\0' ||
16             (de->d_name[1] == '.' && de->d_name[2] == '\0')))
17             continue;
18         strcpy(filename, de->d_name);
19         openDevice(devname);
20     }
21     closedir(dir);
22     return 0;
23 }
```

第7行代码首先调用函数opendir来打开输入设备目录/dev/input，接着第13行到第20行的while循环调用函数readdir来遍历它里面的每一个输入设备文件，并且调用成员函数openDevice来打开它们。

1　inotify是Linux 2.6.13 内核中新引入的一种文件系统变化通知机制，它用来监控文件系统，例如文件创建、删除、读和写等操作。

Step 6：EventHub.openDevice

我们分段来阅读EventHub类的成员函数openDevice。

frameworks/base/libs/ui/EventHub.cpp

```
001 int EventHub::openDevice(const char *deviceName) {
002     int version;
003     int fd;
004     struct pollfd *new_mFDs;
005     device_t **new_devices;
006     char **new_device_names;
007     char name[80];
008     char location[80];
009     char idstr[80];
010     struct input_id id;
011     ......
012
013     AutoMutex _l(mLock);
014
015     fd = open(deviceName, O_RDWR);
016     ......
017
018     name[sizeof(name) - 1] = '\0';
019     ......
020     if(ioctl(fd, EVIOCGNAME(sizeof(name) - 1), &name) < 1) {
021         //fprintf(stderr, "could not get device name for %s, %s\n", deviceName, strerror(errno));
022         name[0] = '\0';
023     }
```

第15行代码首先调用函数open来打开输入设备文件deviceName，并且把得到的文件描述符保存在变量fd中；接着第20行代码使用IO控制命令EVIOCGNAME获得前面所打开的输入设备的名称，并且保存在变量name中。

frameworks/base/libs/ui/EventHub.cpp

```
024     int devid = 0;
025     while (devid < mNumDevicesById) {
026         if (mDevicesById[devid].device == NULL) {
027             break;
028         }
029         devid++;
030     }
031     if (devid >= mNumDevicesById) {
032         device_ent* new_devids = (device_ent*)realloc(mDevicesById,
033                 sizeof(mDevicesById[0]) * (devid + 1));
034         if (new_devids == NULL) {
035             LOGE("out of memory");
036             return -1;
037         }
038         mDevicesById = new_devids;
039         mNumDevicesById = devid+1;
040         mDevicesById[devid].device = NULL;
041         mDevicesById[devid].seq = 0;
042     }
043
044     mDevicesById[devid].seq = (mDevicesById[devid].seq+(1<<SEQ_SHIFT))&SEQ_MASK;
045     if (mDevicesById[devid].seq == 0) {
046         mDevicesById[devid].seq = 1<<SEQ_SHIFT;
047     }
```

EventHub类有一个成员变量mDevicesById，它指向了一个device_ent结构体数组，数组的大小保存在成员变量mNumDevicesById中。这个数组是用来为正在打开的输入设备分配ID的。

结构体device_ent的定义如下所示。

frameworks/base/include/ui/EventHub.h

```
1 struct device_ent {
2     device_t* device;
3     uint32_t seq;
4 };
```

成员变量device指向了一个输入设备，当我们将这个输入设备的ID值的低16位清0之后，就可以得到成员变量seq的值。

我们通过一个具体的例子来说明EventHub类的成员函数openDevice是如何为一个正在打开的输入设备分配ID的。

假设EventHub类开始的时候打开了两个输入设备，那么这两个输入设备的ID值就分别为0x10000和0x10001。这两个输入设备的ID值的低16位值分别等于0和1，表示用来描述它们的一个device_t结构体分别保存在数组mDevicesById的第0个和第1个位置中。此外，这两个输入设备的ID值的高16位值均等于1，表示数组mDevicesById的第0个和第1个位置是第一次被使用的。过了一段时间之后，假设ID值为0x10001的输入设备被关闭了，这时候保存在数组mDevicesById的第1个位置中的device_ent结构体的成员变量device就会被设置为NULL，但是它的成员变量seq的值仍然保持为0x10000（0x10001的低16位清0后得到的值）。又过了一段时间之后，假设又有一个输入设备被打开了，这时候EventHub类的成员函数openDevice会发现保存在数组mDevicesById的第1个位置中的device_ent结构体的成员变量device的值等于NULL，于是它就会将新打开的输入设备的ID值的低16位设置为1，接着再将保存在数组mDevicesById的第1个位置中的device_ent结构体的成员变量seq的值（0x10000）的高16位增加1，并且将结果值（0x20000）的高16位（2）作为新打开的输入设备的ID值的高16位。最后就可以得到新打开的输入设备的ID值为0x20001。

结合这个例子，我们就可以很好地理解第24行到第44行代码了。接下来第45行的if语句判断保存在数组mDevicesById的第devid个位置中的device_ent结构体的成员变量seq的值是否等于0。如果等于0，那么就说明这个位置被重复使用的次数已经大于0xffff次了。在这种情况下，第46行代码就会将它的值重置为1，以便可以开始新一轮的计数。

frameworks/base/libs/ui/EventHub.cpp

```
048     new_mFDs = (pollfd*)realloc(mFDs, sizeof(mFDs[0]) * (mFDCount + 1));
049     new_devices = (device_t**)realloc(mDevices, sizeof(mDevices[0]) * (mFDCount + 1));
050     ......
051     mFDs = new_mFDs;
052     mDevices = new_devices;
053     ......
054
055     device_t* device = new device_t(devid|mDevicesById[devid].seq, deviceName, name);
056     ......
057
058     device->fd = fd;
059     mFDs[mFDCount].fd = fd;
060     mFDs[mFDCount].events = POLLIN;
061     mFDs[mFDCount].revents = 0;
```

由于前面打开了一个新的输入设备，因此，第48行到第52行代码就分别在数组mFDs和mDevices的末尾新增一个位置，以便可以用来保存新打开的输入设备的信息。

第55行代码创建了一个device_t结构体device，并且将它的成员变量fd的值设置为新打开的输入设备的文件描述符，接着第59行代码又将新打开的输入设备的文件描述符保存在数组mFDs的最后一个位置中的pollfd结构体的成员变量fd中。由于我们需要监控新打开的输入设备的IO输入事件，因此，第60行代码还会将保存在数组mFDs的最后一个位置中的pollfd结构体的成员变量events的值设置为POLLIN。

frameworks/base/libs/ui/EventHub.cpp

```
062     // Figure out the kinds of events the device reports.
063     uint8_t key_bitmask[sizeof_bit_array(KEY_MAX + 1)];
064     memset(key_bitmask, 0, sizeof(key_bitmask));
065
066     LOGV("Getting keys...");
067     if (ioctl(fd, EVIOCGBIT(EV_KEY, sizeof(key_bitmask)), key_bitmask) >= 0) {
068         // See if this is a keyboard.  Ignore everything in the button range except for
069         // gamepads which are also considered keyboards.
070         if (containsNonZeroByte(key_bitmask, 0, sizeof_bit_array(BTN_MISC))
071             || containsNonZeroByte(key_bitmask, sizeof_bit_array(BTN_GAMEPAD),
072             sizeof_bit_array(BTN_DIGI))
073             || containsNonZeroByte(key_bitmask, sizeof_bit_array(KEY_OK),
074             sizeof_bit_array(KEY_MAX + 1))) {
075                 device->classes |= INPUT_DEVICE_CLASS_KEYBOARD;
076                 ......
077         }
078     }
```

第67行和第70行的两个if语句用来判断新打开的输入设备是否是一个键盘设备。如果是，那么第75行代码就会将前面所创建的一个device_t结构体device的成员变量classes的第INPUT_DEVICE_CLASS_KEYBOARD位的值设置为1。

frameworks/base/libs/ui/EventHub.cpp

```
079     if ((device->classes & INPUT_DEVICE_CLASS_KEYBOARD) != 0) {
080         char tmpfn[sizeof(name)];
081         char keylayoutFilename[300];
082
083         // a more descriptive name
084         device->name = name;
085
086         // replace all the spaces with underscores
087         strcpy(tmpfn, name);
088         for (char *p = strchr(tmpfn, ' '); p && *p; p = strchr(tmpfn, ' '))
089             *p = '_';
090
091         // find the .kl file we need for this device
092         const char* root = getenv("ANDROID_ROOT");
093         snprintf(keylayoutFilename, sizeof(keylayoutFilename),
094             "%s/usr/keylayout/%s.kl", root, tmpfn);
095         bool defaultKeymap = false;
096         if (access(keylayoutFilename, R_OK)) {
097             snprintf(keylayoutFilename, sizeof(keylayoutFilename),
098                 "%s/usr/keylayout/%s", root, "qwerty.kl");
099             defaultKeymap = true;
100         }
101         status_t status = device->layoutMap->load(keylayoutFilename);
102         ......
103
104         // tell the world about the devname (the descriptive name)
105         if (!mHaveFirstKeyboard && !defaultKeymap && strstr(name, "-keypad")) {
106             // the built-in keyboard has a well-known device ID of 0,
107             // this device better not go away.
108             mHaveFirstKeyboard = true;
109             mFirstKeyboardId = device->id;
110             ......
111         } else {
112             // ensure mFirstKeyboardId is set to -something-.
113             if (mFirstKeyboardId == 0) {
114                 mFirstKeyboardId = device->id;
115             }
116         }
117         ......
118     }
```

第79行的if语句继续检查新打开的输入设备是否是一个键盘设备。如果是，那么接下来第80行到第116行代码就主要用来为它加载一个键盘布局文件。有了这个键盘布局文件之后，我们就可以将这个键盘设备的扫描码转换为键盘码了。

第87行代码首先将新打开的键盘设备的名称复制到一个临时字符数组tmpfn中，接着第88行和第89行的while循环再使用下横线“_”来替换掉这个临时字符数组中的空格字符。

第92行代码首先获得环境变量ANDROID_ROOT的值，并且保存在字符串变量root中。环境变量ANDROID_ROOT是用来描述Android系统的根目录的，它的值一般就等于“/system”。第93行和第94行代码使用字符串root以及临时字符数组tmpfn来格式化字符串“%s/usr/keylayout/%s.kl”，并且将结果保存在字符数组keylayoutFilename中，用来作为新打开的键盘设备的键盘布局文件的名称。假设新打开的键盘设备的名称为“abc”，并且环境变量ANDROID_ROOT的值为“/system”，那么这里得到的键盘布局文件的名称就是“/system/usr/keylayout/abc.kl”了。

第96行的if语句调用函数access来检查前面所得到的键盘布局文件是否存在。如果不存在，那么接下来第97行和第98行代码就会将新打开的键盘设备所使用的键盘布局文件的名称设置为“/system/usr/keylayout/qwerty.kl”，并且第99行代码会将变量defaultKeymap的值设置为true，表示新打开的键盘设备所使用的键盘布局文件是系统默认的键盘布局文件。

获得了新打开的键盘设备的键盘布局文件的名称之后，接着第101行代码就调用前面所创建的一个device_t结构体device内部的一个KeyLayoutMap对象的成员函数load将该文件加载到内存中。

第105行的if语句判断新打开的键盘设备是否是一个内建的键盘设备。如果是，那么接下来第108行代码就会将EventHub类的成员变量mHaveFirstKeyboard的值设置为true，并且第109行代码会将新打开的键盘设备的ID值保存在EventHub类的成员变量mFirstKeyboardId中。这样以后当ID值为mFirstKeyboardId的输入设备发生IO输入事件时，EventHub类就可以知道这个输入设备实际上是一个内建的键盘设备。

如果新打开的键盘设备不是一个内建的键盘设备，但是它是到目前为止第一个打开的键盘设备，即第105行的if语句为false，并且第113行的if语句为true，那么接下来第114行代码就会将它的ID值保存在EventHub类的成员变量mFirstKeyboardId中，相当于将它看作是内建的键盘设备。

frameworks/base/libs/ui/EventHub.cpp

```
119     mDevicesById[devid].device = device;
120     ......
121     mDevices[mFDCount] = device;
122
123     mFDCount++;
124     return 0;
125 }
```

第119行和第121行代码分别将前面所创建的一个device_t结构体device保存在数组mDevicesById和mDevices中。最后第123行代码将已经打开的输入设备的个数mFDCount增加1。

这一步执行完成之后，沿着调用路径返回到前面的Step 3中，即EventHub类的成员函数getEvent中，接下来这个函数就会将EventHub类的成员变量mOpened的值设置为true，表示系统中的所有输入设备都已经打开了。

至此，InputReader的启动过程就分析完成了，整个InputManager的启动过程也分析完成了。接下来我们继续分析应用程序窗口与InputManager建立连接的过程，即InputChannel的注册过程。

14.3 InputChannel的注册过程

我们知道，在Android系统中，一个应用程序窗口是由一个Activity组件来描述的。每一个Activity组件都有一个关联的ViewRoot对象。一个ViewRoot对象除了负责绘制与它所关联的Activity组件所描述的应用程序窗口之外，还负责接收InputDispatcher分发过来的键盘事件。

一个Activity组件和一个ViewRoot对象的关系如图14-8所示。

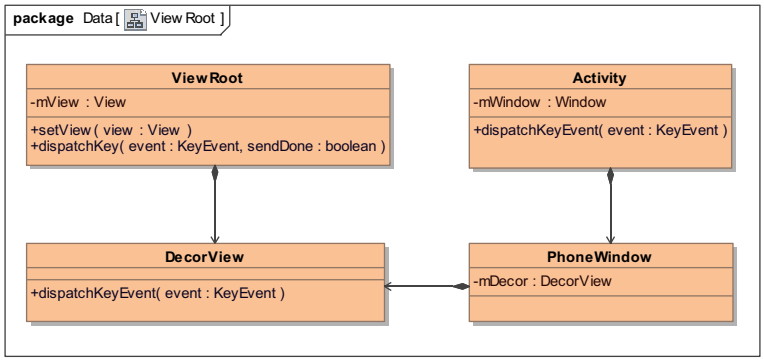


图14-8 Activity组件和ViewRoot对象的关系

每一个Activity组件的内部都包含有一个PhoneWindow对象，保存在它的成员变量mWindow中；每一个PhoneWindow对象的内部又包含有一个DecorView对象，保存在它的成员变量mDecor中；每一个DecorView对象最终又会被设置到一个ViewRoot对象的成员变量mView中，这是通过调用ViewRoot类的成员函数setView来实现的。

一个Activity组件在启动时，会通过与它所关联的ViewRoot对象和InputManager建立一个连接。当系统发生一个键盘事件时，InputManager内部的InputDispatcher就会将这个键盘事件分发给与当前激活的Activity组件所关联的ViewRoot对象处理。一个ViewRoot对象接收到InputDispatcher分发过来的一个键盘事件之后，就会调用它的成员函数dispatchKey来处理；这个ViewRoot对象的成员函数dispatchKey接着又会将接收到的键盘事件交给它的成员变量mView所指向的一个DecorView对象的成员函数dispatchKeyEvent来处理。一个DecorView对象接收到一个ViewRoot对象分发过来的一个键盘事件之后，接下来就会找到包含它的一个PhoneWindow对象，以便可以将接收到的键盘事件分发给包含了该PhoneWindow对象的Activity组件的成员函数dispatchKeyEvent来处理。

从上面的描述就可以看出，一个Activity组件如果想要接收到系统的键盘事件，那么它就必须主动与InputManager建立一个连接。从前面14.1小节的内容可以知道，这个连接是通过一个Server端InputChannel和一个Client端InputChannel来描述的。相应地，Activity组件与InputManager建立连接的过程又称为注册InputChannel的过程，其中，Server端InputChannel注册在InputManager中，而Client端InputChannel注册在Activity组件所运行的应用程序进程的主线程中。在注册Server端InputChannel的过程中，InputManager又会将系统当前激活的Activity组件，即当前激活的应用程序窗口注册到InputDispatcher中，以便InputDispatcher可以将键盘事件分发给它处理。

下面我们首先分析Server端InputChannel和Client端InputChannel的创建过程，然后分析Server端InputChannel的注册过程，以及系统当前激活的应用程序窗口的注册过程，最后分析Client端InputChannel的注册过程。

14.3.1　创建InputChannel

前面提到，一个Activity组件在启动时，会将它内部的一个DecorView对象设置到与它所关联的一个ViewRoot对象的内部。在设置的过程中，这个ViewRoot对象会创建一个Server端InputChannel和一个Client端InputChannel，以便可以与InputManager建立一个接收键盘事件的连接。

将一个DecorView对象设置到一个ViewRoot对象内部是通过调用ViewRoot类的成员函数setView来实现的，接下来我们就从这个函数开始分析一个Server端InputChannel和一个Client端InputChannel的创建过程，如图14-9所示。

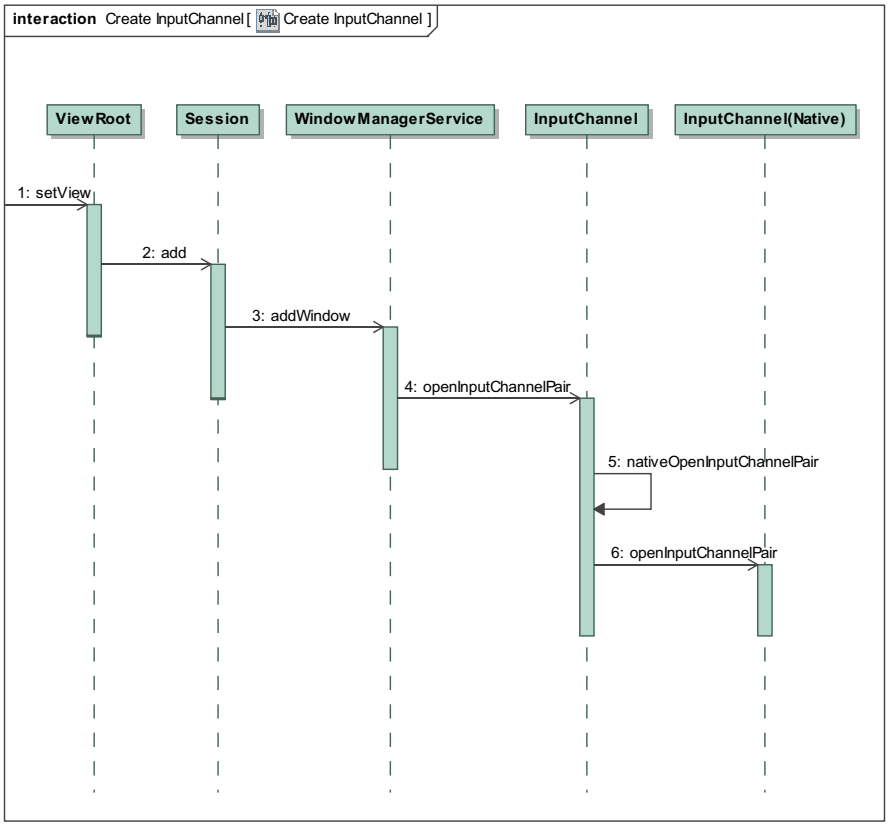


图14-9　InputChannel的创建过程

这个过程一共分为6个步骤，下面我们就详细分析每一个步骤。

Step 1：ViewRoot.setView

frameworks/base/core/java/android/view/ViewRoot.java

```
01 public final class ViewRoot extends Handler implements ViewParent,
02         View.AttachInfo.Callbacks {
03     ......
04
05     public void setView(View view, WindowManager.LayoutParams attrs,
06             View panelParentView) {
07         ......
08
09         synchronized (this) {
10             if (mView == null) {
11                 mView = view;
12                 ......
```

```
13
14                  mAdded = true;
15                  ......
16
17                  mInputChannel = new InputChannel();
18                  try {
19                      res = sWindowSession.add(mWindow, mWindowAttributes,
20                          getHostVisibility(), mAttachInfo.mContentInsets,
21                          mInputChannel);
22                  } catch (RemoteException e) {
23                      ......
24                  } finally {
25                      ......
26                  }
27                  ......
28
29                  if (view instanceof RootViewSurfaceTaker) {
30                      mInputQueueCallback =
31                          ((RootViewSurfaceTaker)view).willYouTakeTheInputQueue();
32                  }
33                  if (mInputQueueCallback != null) {
34                      mInputQueue = new InputQueue(mInputChannel);
35                      mInputQueueCallback.onInputQueueCreated(mInputQueue);
36                  } else {
37                      InputQueue.registerInputChannel(mInputChannel, mInputHandler,
38                          Looper.myQueue());
39                  }
40
41                  ......
42              }
43          }
44      }
45
46      ......
47  }
```

第10行的if语句首先检查ViewRoot类的成员变量mView是否等于null。如果等于null，那么就说明它还没有与一个DecorView对象关联起来。因此，接下来第11行代码就会将参数view所描述的一个DecorView对象保存在它的成员变量mView中，并且第14行代码将它的成员变量mAdded的值设置为true，表示它已经与一个DecorView对象关联起来了。

第17行代码创建了一个InputChannel，并且保存在ViewRoot类的成员变量mInputChannel中，用来描述一个Client端InputChannel。

注意

这个Client端InputChannel还没有被初始化，即它还不具有与InputManager建立连接的功能。

第19行到第21行代码调用ViewRoot类的静态成员变量sWindowSession的成员函数add将正在启动的应用程序窗口添加到Window管理服务WindowManagerService中。在这个过程中，Window管理服务WindowManagerService会完成三件事情。第一件事情是创建一对InputChannel，其中一个用作Server端InputChannel，另外一个用来初始化前面所创建的一个Client端InputChannel；第二件事情是将前面所创建的一个Server端InputChannel注册到InputManager中；第三件事情是将正在添加的应用程序窗口作为系统当前激活的应用程序窗口，并且注册到InputDispatcher中。

第29行的if语句检查参数view指向的是否是一个RootViewSurfaceTaker对象。如果是，那么ViewRoot类接下来就不会负责将前面所创建的一个Client端InputChannel注册到正在启动的Activity组件所运行在的应用程序进程的主线程中。为了方便描述，我们假设参数view指向的是一个DecorView对象，

即第29行的if语句为false，那么接下来第37行和第38行代码就会调用InputQueue类的静态成员函数registerInputChannel将前面所创建的一个Client端InputChannel注册到正在启动的Activity组件所运行在的应用程序进程的主线程中。

本节主要关注InputChannel的创建过程，因此，接下来我们就继续分析ViewRoot类的静态成员变量sWindowSession的成员函数add的实现。

ViewRoot类的静态成员变量sWindowSession是一个Binder代理对象，它引用了运行在Window管理服务WindowManagerService中的一个类型为Session的Binder本地对象，它的创建过程如下所示。

frameworks/base/core/java/android/view/ViewRoot.java

```
01 public final class ViewRoot extends Handler implements ViewParent,
02         View.AttachInfo.Callbacks {
03     ......
04
05     static IWindowSession sWindowSession;
06     ......
07
08     public static IWindowSession getWindowSession(Looper mainLooper) {
09         synchronized (mStaticInit) {
10             if (!mInitialized) {
11                 try {
12                     InputMethodManager imm = InputMethodManager.getInstance(mainLooper);
13                     sWindowSession = IWindowManager.Stub.asInterface(
14                             ServiceManager.getService("window"))
15                             .openSession(imm.getClient(), imm.getInputContext());
16                     mInitialized = true;
17                 } catch (RemoteException e) {
18                 }
19             }
20             return sWindowSession;
21         }
22     }
23
24     ......
25 }
```

第13行到第15行代码首先调用ServiceManager类的静态成员函数getService来获得一个WindowManagerService代理对象，然后调用这个WindowManagerService代理对象的成员函数openSession来获得一个Session代理对象，最后再将这个Session代理对象保存在ViewRoot类的静态成员变量sWindowSession中。

回到ViewRoot类的成员函数setView中，接下来它就会调用ViewRoot类的静态成员变量sWindowSession的成员函数add向运行在Window管理服务WindowManagerService中的一个Session对象发送一个进程间通信请求，以便可以将正在启动的应用程序窗口添加到Window管理服务WindowManagerService中。这个进程间通信请求是由在WindowManagerService类内部定义的Session类的成员函数add来处理的，接下来我们就分析它的实现。

Step 2：Session.add

frameworks/base/services/java/com/android/server/WindowManagerService.java

```
01 public class WindowManagerService extends IWindowManager.Stub
02         implements Watchdog.Monitor {
03     ......
04
05     private final class Session extends IWindowSession.Stub
06             implements IBinder.DeathRecipient {
07         ......
08
```

```
09          public int add(IWindow window, WindowManager.LayoutParams attrs,
10                  int viewVisibility, Rect outContentInsets, InputChannel outInputChannel) {
11              return addWindow(this, window, attrs, viewVisibility, outContentInsets,
12                  outInputChannel);
13          }
14
15          ......
16      }
17
18      ......
19  }
```

第11行和第12行代码调用WindowManagerService类的成员函数addWindow将正在启动的应用程序窗口添加到Window管理服务WindowManagerService中。

Step 3：WindowManagerService.addWindow

frameworks/base/services/java/com/android/server/WindowManagerService.java

```
01 public class WindowManagerService extends IWindowManager.Stub
02          implements Watchdog.Monitor {
03      ......
04
05      final HashMap<IBinder, WindowState> mWindowMap = new HashMap<IBinder, WindowState>();
06      ......
07
08      final ArrayList<WindowState> mWindows = new ArrayList<WindowState>();
09      ......
10
11      public int addWindow(Session session, IWindow client,
12              WindowManager.LayoutParams attrs, int viewVisibility,
13              Rect outContentInsets, InputChannel outInputChannel) {
14          ......
15          WindowState win = null;
16
17          synchronized(mWindowMap) {
18              ......
19
20              win = new WindowState(session, client, token,
21                  attachedWindow, attrs, viewVisibility);
22              ......
23
24              if (outInputChannel != null) {
25                  String name = win.makeInputChannelName();
26                  InputChannel[] inputChannels = InputChannel.openInputChannelPair(name);
27                  win.mInputChannel = inputChannels[0];
28                  inputChannels[1].transferToBinderOutParameter(outInputChannel);
29                  mInputManager.registerInputChannel(win.mInputChannel);
30              }
31              ......
32
33              mWindowMap.put(client.asBinder(), win);
34              ......
35
36              if (attrs.type == TYPE_INPUT_METHOD) {
37                  ......
38              } else if (attrs.type == TYPE_INPUT_METHOD_DIALOG) {
39                  ......
40              } else {
41                  addWindowToListInOrderLocked(win, true);
42                  ......
```

```
43              }
44              ......
45
46              boolean focusChanged = false;
47              if (win.canReceiveKeys()) {
48                  focusChanged = updateFocusedWindowLocked(UPDATE_FOCUS_WILL_ASSIGN_LAYERS);
49                  ......
50              }
51              ......
52
53              if (focusChanged) {
54                  finishUpdateFocusedWindowAfterAssignLayersLocked();
55              }
56
57              ......
58          }
59
60          ......
61      }
62
63      ......
64  }
```

第20行和第21行代码首先创建了一个WindowState对象win来描述正在启动的应用程序窗口。

参数outInputChannel指向了在前面的Step 1中所创建的一个Client端InputChannel，它的值不等于null，即第24行的if语句为true。因此，接下来第26行代码就会调用InputChannel类的静态成员函数openInputChannelPair来创建一对InputChannel，并且保存在数组inputChannels中。

数组inputChannels的第一个元素保存的是一个Server端InputChannel，第27行代码将它保存在前面所创建的一个WindowState对象win的成员变量mInputChannel中。数组inputChannels的第二个元素保存的是一个Client端InputChannel，第28行代码使用它来初始化参数outInputChannel所描述的一个Client端InputChannel，即在前面的Step 1中所创建的一个Client端InputChannel。

从前面14.2.1小节的Step 1可以知道，Window管理服务WindowManagerService的成员变量mInputManager指向了系统的输入管理器InputManager，第29行代码调用它的成员函数registerInputChannel来注册前面所创建的一个Server端 InputChannel。

参数client指向了一个实现了IWindow接口的Binder代理对象，它引用了ViewRoot类中的成员变量mWindow所描述的一个Binder本地对象。Window管理服务就是通过这个Binder本地对象来管理运行在应用程序进程中的应用程序窗口的，例如，设置它的大小以及可见性等。第33行代码首先获得参数client的一个IBinder接口，接着再以这个IBinder接口为关键字，将前面所创建的WindowState对象win保存在Window管理服务WindowManagerService的成员变量mWindowMap所描述的一个HashMap中。

第36行和第38行的if语句检查正在启动的应用程序窗口是否是一个与输入法相关的窗口。如果不是，那么接下来第41行代码就会调用Window管理服务WindowManagerService的成员函数addWindowToListInOrderLocked将前面所创建的WindowState对象win保存在Window管理服务WindowManagerService的成员变量mWindows所描述的一个应用程序窗口列表中。

注意

Window管理服务WindowManagerService的成员变量mWindows所保存的应用程序窗口都是需要接收键盘事件的，不过在同一时刻，只有系统当前激活的应用程序窗口才可以接收到。

至此，Window管理服务WindowManagerService就把正在启动的应用程序窗口添加到内部了，分别保存在其内部的两个成员变量mWindowMap和mWindows中。

第47行的if语句检查正在启动的应用程序窗口是否可以接收键盘事件。如果可以，那么接下来第48行代码就会调用Window管理服务WindowManagerService的成员函数updateFocusedWindowLocked将它作为系统当前激活的应用程序窗口，即将前面所创建的WindowState对象win保存在Window管理服务WindowManagerService的成员变量mCurrentFocus中。

由于正在启动的应用程序窗口是一个新的应用程序窗口，它与系统当前激活的应用程序窗口不是同一个应用程序窗口，因此，Window管理服务WindowManagerService的成员函数updateFocusedWindowLocked将它作为系统当前激活的应用程序窗口之后，就会返回一个true给调用者，以便调用者可以知道系统当前激活的应用程序窗口发生改变了。这时候第54行的代码就会调用Window管理服务WindowManagerService的成员函数finishUpdateFocusedWindowAfterAssignLayersLocked将正在启动的应用程序窗口注册到InputDispatcher中，以便InputDispatcher可以将键盘事件分发给它处理。

本节主要关注InputChannel的创建过程，因此，接下来我们就继续分析InputChannel类的静态成员函数openInputChannelPair的实现，以便可以了解它是如何创建一个Server端InputChannel和一个Client端InputChannel的。

Step 4：InputChannel.openInputChannelPair

frameworks/base/core/java/android/view/InputChannel.java

```
01 public final class InputChannel implements Parcelable {
02     ......
03
04     private static native InputChannel[] nativeOpenInputChannelPair(String name);
05     ......
06
07     public static InputChannel[] openInputChannelPair(String name) {
08         ......
09
10         return nativeOpenInputChannelPair(name);
11     }
12
13     ......
14 }
```

第10行代码调用了InputChannel类的静态成员函数nativeOpenInputChannelPair来进一步创建一个Server端InputChannel和一个Client端InputChannel。

Step 5：InputChannel.nativeOpenInputChannelPair

frameworks/base/core/jni/android_view_InputChannel.cpp

```
01 static jobjectArray android_view_InputChannel_nativeOpenInputChannelPair(JNIEnv* env,
02         jclass clazz, jstring nameObj) {
03      const char* nameChars = env->GetStringUTFChars(nameObj, NULL);
04      String8 name(nameChars);
05      env->ReleaseStringUTFChars(nameObj, nameChars);
06
07      sp<InputChannel> serverChannel;
08      sp<InputChannel> clientChannel;
09      status_t result = InputChannel::openInputChannelPair(name, serverChannel, clientChannel);
10      ......
11
12      // TODO more robust error checking
13      jobject serverChannelObj = android_view_InputChannel_createInputChannel(env,
14          new NativeInputChannel(serverChannel));
15      jobject clientChannelObj = android_view_InputChannel_createInputChannel(env,
```

```
16          new NativeInputChannel(clientChannel));
17
18      jobjectArray channelPair = env->NewObjectArray(2, gInputChannelClassInfo.clazz, NULL);
19      env->SetObjectArrayElement(channelPair, 0, serverChannelObj);
20      env->SetObjectArrayElement(channelPair, 1, clientChannelObj);
21      return channelPair;
22 }
```

InputChannel类的静态成员函数nativeOpenInputChannelPair是一个JIN方法，它是由C++层中的函数android_view_InputChannel_nativeOpenInputChannelPair来实现的。

第9行代码首先调用C++层的InputChannel类的静态成员函数openInputChannelPair来创建一个C++层的Server端InputChannel和一个C++层的Client端InputChannel，然后第13行到第16行代码调用函数android_view_InputChannel_createInputChannel分别将它们封装成一个Java层的Server端InputChannel和一个Java层的Client端InputChannel，最后第18行到第21行代码再将这两个Java层的InputChannel保存在一个Java数组channelPair中返回给调用者。

接下来，我们继续分析C++层的InputChannel类的静态成员函数openInputChannelPair的实现，以便可以了解它是如何在C++层中创建一个Server端InputChannel和一个Client端InputChannel的。

Step 6：InputChannel.openInputChannelPair

frameworks/base/libs/ui/InputTransport.cpp

```
01 status_t InputChannel::openInputChannelPair(const String8& name,
02         sp<InputChannel>& outServerChannel, sp<InputChannel>& outClientChannel) {
03     status_t result;
04
05     int serverAshmemFd = ashmem_create_region(name.string(), DEFAULT_MESSAGE_BUFFER_SIZE);
06     if (serverAshmemFd < 0) {
07         ......
08     } else {
09         result = ashmem_set_prot_region(serverAshmemFd, PROT_READ | PROT_WRITE);
10         if (result < 0) {
11             ......
12         } else {
13             // Dup the file descriptor because the server and client input channel objects that
14             // are returned may have different lifetimes but they share the same shared memory region.
15             int clientAshmemFd;
16             clientAshmemFd = dup(serverAshmemFd);
17             if (clientAshmemFd < 0) {
18                 ......
19             } else {
20                 int forward[2];
21                 if (pipe(forward)) {
22                     ......
23                 } else {
24                     int reverse[2];
25                     if (pipe(reverse)) {
26                         ......
27                     } else {
28                         String8 serverChannelName = name;
29                         serverChannelName.append(" (server)");
30                         outServerChannel = new InputChannel(serverChannelName,
31                             serverAshmemFd, reverse[0], forward[1]);
32
33                         String8 clientChannelName = name;
34                         clientChannelName.append("(client)");
35                         outClientChannel = new InputChannel(clientChannelName,
```

```
36                              clientAshmemFd, forward[0], reverse[1]);
37                          return OK;
38                      }
39                      ......
40                  }
41                  ......
42              }
43          }
44      }
45      ......
46 }
```

在分析C++层的InputChannel类的静态成员函数openInputChannelPair的实现之前，我们首先了解创建一个C++层的InputChannel需要哪些参数。

C++层的InputChannel类的构造函数的实现如下所示。

frameworks/base/libs/ui/InputTransport.cpp

```
1 InputChannel::InputChannel(const String8& name, int32_t ashmemFd, int32_t receivePipeFd,
2     int32_t sendPipeFd) :
3     mName(name), mAshmemFd(ashmemFd), mReceivePipeFd(receivePipeFd), mSendPipeFd(sendPipeFd) {
4     ......
5 }
```

从这里就可以看出，创建一个C++层的InputChannel需要四个参数。第一个参数name用来描述要创建的InputChannel的名称；第二个参数ashmemFd是一个文件描述符，它指向了一块匿名共享内存，这块匿名共享内存是用来传递键盘事件数据的；第三个参数receivePipeFd和第四个参数sendPipeFd分别指向了两个不同管道的读端文件描述符和写端文件描述符，这两个管道是用来在应用程序窗口与InputManager之间建立交叉连接的。

回到InputChannel类的静态成员函数openInputChannelPair中，参数name就是用来描述即将要创建的InputChannel的名称的。第29行和第34行代码分别在它的后面附加一个“ (server)”和一个“ (clinet)”字符串，就可以得到即将要创建的Server端InputChannel和Client端InputChannel的名称。

获得了即将要创建的Server端InputChannel和Client端InputChannel的名称之后，我们还需要一块匿名共享内存以及两个不同的管道，以便可以获得三个文件描述符。

第5行代码调用运行时库cutils提供的匿名共享内存创建接口ashmem_create_region来创建一块匿名共享内存，它的大小等于DEFAULT_MESSAGE_BUFFER_SIZE。DEFAULT_MESSAGE_BUFFER_SIZE是一个宏，它的值被定义为16384，因此，第5行代码创建的匿名共享内存块的大小就为16KB。

前面所创建的匿名共享内存分别使用两个文件描述符来描述，其中一个是serverAshmemFd，另外一个是clientAshmemFd，它们分别用来作为即将要创建的Server端InputChannel和Client端InputChannel的匿名共享内存文件描述符。

第21行代码调用函数pipe来创建一个前向管道，它的读端文件描述符和写端文件描述符分别保存在数组forward的第一个元素和第二个元素中；接着第25行代码又继续调用函数pipe来创建一个反向管道，它的读端文件描述符和写端文件描述符分别保存在数组reverse的第一个元素和第二个元素中。

至此，创建一个Server端InputChannel和一个Client端InputChannel所需要的参数就都准备好了。接下来第30行和第31行代码就以serverChannelName、serverAshmemFd、reverse [0]和forward[1]为参数来创建一个Server端InputChannel，而第35行和第36行代码以clientChannelName、clientAshmemFd、forward[0]和reverse[1]来创建一个Client端InputChannel。

从Server端InputChannel和Client端InputChannel的创建过程就可以看出，它们是通过一个前向管道

和一个反向管道交叉连接在一起的，并且在内部引用了同一块匿名共享内存。在后面的14.4小节中分析键盘消息的分发过程时，我们就会看到这两个管道以及这块匿名共享内存的作用了。

这一步执行完成之后，一对交叉连接在一起的Server端InputChannel和Client端InputChannel就创建完成了。返回到前面的Step 3中，即WindowManagerService类的成员函数addWindow中，接下来我们继续分析Server端InputChannel的注册过程。

14.3.2　注册Server端InputChannel

从前面14.3.1小节的Step 3可以知道，Java层的InputManager类的成员函数registerInputChannel是用来注册一个Server端InputChannel的，接下来我们就从这个函数开始分析一个Server端InputChannel的注册过程，如图14-10所示。

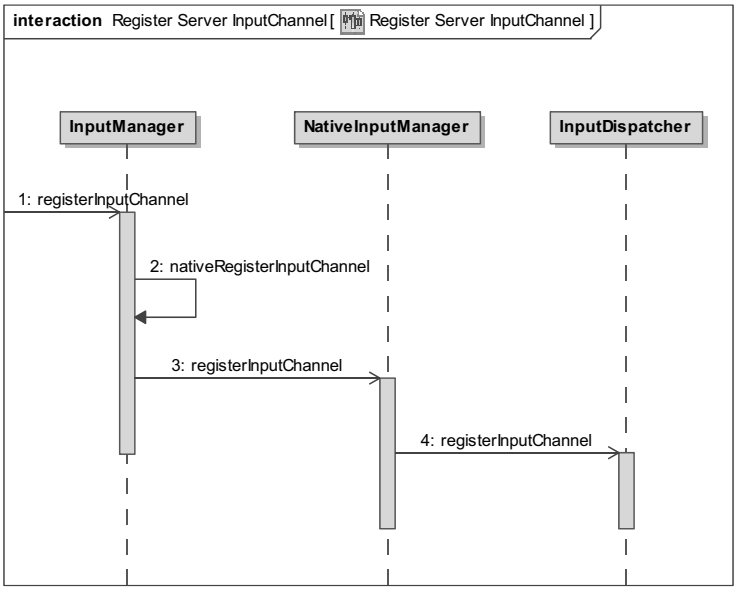


图14-10　Server端InputChannel的注册过程

这个过程一共分为4个步骤，下面我们就详细分析每一个步骤。

Step 1：InputManager.registerInputChannel

frameworks/base/services/java/com/android/server/InputManager.java

```
01 public class InputManager {
02     ......
03
04     private static native void nativeUnregisterInputChannel(InputChannel inputChannel);
05     ......
06
07     public void registerInputChannel(InputChannel inputChannel) {
08         if (inputChannel == null) {
09             throw new IllegalArgumentException("inputChannel must not be null.");
10         }
11
12         nativeRegisterInputChannel(inputChannel, false);
13     }
14
15     ......
16 }
```

第12行代码调用了InputManager类的成员函数nativeRegisterInputChannel来注册一个Server端InputChannel。

Step 2：InputManager.nativeRegisterInputChannel

frameworks/base/services/jni/com_android_server_InputManager.cpp

```
01 static void android_server_InputManager_nativeRegisterInputChannel(JNIEnv* env, jclass
clazz,
02         jobject inputChannelObj, jboolean monitor) {
03     ......
04
05     sp<InputChannel> inputChannel = android_view_InputChannel_getInputChannel(env,
06        inputChannelObj);
07     ......
08
09     status_t status = gNativeInputManager->registerInputChannel(
10        env, inputChannel, inputChannelObj, monitor);
11
12     ......
13 }
```

InputManager类的成员函数nativeRegisterInputChannel是一个JNI方法，它是由C++层的函数android_server_InputManager_nativeRegisterInputChannel来实现的。

参数inputChannelObj指向了一个Java层的InputChannel，用来描述即将要注册的一个Server端InputChannel，第5行和第6行代码调用函数android_view_InputChannel_getInputChannel将它转换为一个C++层的InputChannel。这个C++层的InputChannel是在前面14.3.1小节的Step 5中创建的，创建完成之后就被调用函数android_view_InputChannel_createInputChannel转换为一个Java层的InputChannel返回给调用者。

从前面14.2.1小节的Step 3可以知道，全局变量gNativeInputManager指向了一个C++层的NativeInputManager对象，第9行和第10行代码调用它的成员函数registerInputChannel来继续执行注册Server端InputChannel的操作。

Step 3：NativeInputManager.registerInputChannel

frameworks/base/services/jni/com_android_server_InputManager.cpp

```
1 status_t NativeInputManager::registerInputChannel(JNIEnv* env,
2         const sp<InputChannel>& inputChannel, jobject inputChannelObj, bool monitor) {
3     ......
4
5     status = mInputManager->getDispatcher()->registerInputChannel(inputChannel, monitor);
6
7     ......
8 }
```

NativeInputManager类的成员变量mInputManager指向了一个C++层的InputManager对象，这个C++层的InputManager对象是在前面14.2.1小节的Step 3中创建的，第5行代码首先获得它内部的一个InputDispatcher对象，接着再调用这个InputDispatcher对象的成员函数registerInputChannel来继续执行注册Server端InputChannel的操作。

Step 4：InputDispatcher.registerInputChannel

frameworks/base/libs/ui/InputDispatcher.cpp

```
01 status_t InputDispatcher::registerInputChannel(const sp<InputChannel>& inputChannel, bool monitor) {
02     ......
```

```
03
04     { // acquire lock
05         AutoMutex _l(mLock);
06
07         if (getConnectionIndexLocked(inputChannel) >= 0) {
08             ......
09             return BAD_VALUE;
10         }
11
12         sp<Connection> connection = new Connection(inputChannel);
13         status_t status = connection->initialize();
14         ......
15
16         int32_t receiveFd = inputChannel->getReceivePipeFd();
17         mConnectionsByReceiveFd.add(receiveFd, connection);
18         ......
19
20         mLooper->addFd(receiveFd, 0, ALOOPER_EVENT_INPUT, handleReceiveCallback, this);
21
22         ......
23     } // release lock
24     return OK;
25 }
```

第7行的if语句首先调用成员函数getConnectionIndexLocked来检查参数inputChannel所描述的一个Server端InputChannel是否已经注册过了。如果已经注册过，那么第9行代码就直接出错返回了。

注册到InputDispatcher中的Server端InputChannel首先会被封装成一个Connection对象，然后再以这个Server端InputChannel内部的一个反向管道的读端文件描述符为关键字，保存在InputDispatcher类的成员变量mConnectionsByReceiveFd中。

第12行代码首先将参数inputChannel所描述的一个Server端InputChannel封装成一个Connection对象，接着第13行代码再调用这个Connection对象的成员函数initialize对它执行一些初始化工作。

Connection类的成员函数initialize的实现如下所示。

frameworks/base/libs/ui/InputDispatcher.cpp

```
1 status_t InputDispatcher::Connection::initialize() {
2     return inputPublisher.initialize();
3 }
```

Connection类的成员变量inputPublisher指向了一个InputPublisher对象，它是用来分发键盘事件给系统当前激活的应用程序窗口处理的，第2行代码调用它的成员函数initialize对它执行一些初始化工作。

InputPublisher类的成员函数initialize的实现如下所示。

frameworks/base/libs/ui/InputTransport.cpp

```
01 status_t InputPublisher::initialize() {
02     ......
03
04     int ashmemFd = mChannel->getAshmemFd();
05     int result = ashmem_get_size_region(ashmemFd);
06     ......
07
08     mAshmemSize = (size_t) result;
09
10     mSharedMessage = static_cast<InputMessage*>(mmap(NULL, mAshmemSize,
11         PROT_READ | PROT_WRITE, MAP_SHARED, ashmemFd, 0));
```

```
12      ......
13
14      return reset();
15  }
```

InputPublisher类的成员变量mChannel指向了一个C++层的Server端InputChannel，第4行代码首先调用它的成员函数getAshmemFd来获得它所引用的一块匿名共享内存的一个文件描述符ashmemFd；接着第5行代码再以文件描述符ashmemFd为参数来调用运行时库cutils所提供的一个匿名共享内存访问接口ashmem_get_size_region，以便可以获得它所描述的一块匿名共享内存的大小；最后第8行代码将这块匿名共享内存的大小保存在InputPublisher类的成员变量mAshmemSize中。

第10行和第11行代码首先调用函数mmap将文件描述符ashmemFd所描述的一块匿名共享内存映射到当前进程的地址空间，即映射到InputDispatcher所运行在的进程的地址空间；接着再将映射后所得到的地址空间内容结构化成一个InputMessage结构体，并且保存在InputPublisher类的成员变量mSharedMessage中。在后面的14.4小节中，我们就会看到，这个InputMessage结构体是用来描述一个即将要分发的键盘事件的数据的。

回到InputDispatcher类的成员函数registerInputChannel中，接下来第16行代码首先获得参数inputChannel所描述的一个Server端InputChannel的内部的一个反向管道的读端文件描述符；接着第17行代码再以这个文件描述符为关键字，将前面所创建的一个Connection对象保存在InputDispatcher类的成员变量mConnectionsByReceiveFd中。

在前面的14.1小节中提到，InputDispatcher类的成员变量mLooper指向了一个C++层的Looper对象，第20行代码调用它的成员函数addFd将前面所注册的一个Server端InputChannel的内部的一个反向管道的读端文件描述符保存在它里面，以便可以监控这个反向管道的IO写事件。当这个反向管道发生了一个IO写事件时，即当这个反向管道被写入了一个新的数据时，InputDispatcher类的静态成员函数handleReceiveCallback就会被调用。

注意

如果这时候InputDispatcher正处于睡眠等待状态，那么它就会首先被唤醒，然后再调用InputDispatcher类的静态成员函数handleReceiveCallback。

在后面的14.4小节中，我们就会看到，当InputDispatcher将一个键盘事件分发给系统当前激活的应用程序窗口之后，它就会进入睡眠等待状态，直到系统当前激活的应用程序窗口唤醒它为止。系统当前激活的应用程序窗口在处理完成一个键盘事件之后，就会向与其关联的一个Client端InputChannel的内部的一个反向管道写入一个新的数据，以便可以将InputDispatcher唤醒，使得它可以继续分发下一个键盘事件。

在前面的13.1小节中提到，一个Looper对象除了用来监控它内部的一个管道的读端文件描述符的IO事件之外，还可以同时监控外部的文件描述符的IO事件，这是通过调用这个Looper对象的成员函数addFd来实现的。在调用一个Looper对象的成员函数addFd来监控一个外部的文件描述符的IO事件时，还需要指定一个关联的回调函数。这个回调函数是在与它所关联的文件描述符发生IO事件时被调用的，这样我们就可以在一个线程消息循环中实现IO事件侦听的功能。

这一步执行完成之后，一个Server端InputChannel就注册完成了。返回到前面14.3.1小节的Step 3中，即WindowManagerService类的成员函数addWindow中，接下来我们继续分析系统当前激活的应用程序窗口的注册过程。

14.3.3　注册系统当前激活的应用程序窗口

从前面14.3.1小节的Step 3可以知道，Window管理服务WindowManagerService的成员函数finishUpdateFocusedWindowAfterAssignLayersLocked是用来注册系统当前激活的应用程序窗口的，接下来我们就从这个函数开始分析系统当前激活的应用程序窗口的注册过程，如图14-11所示。

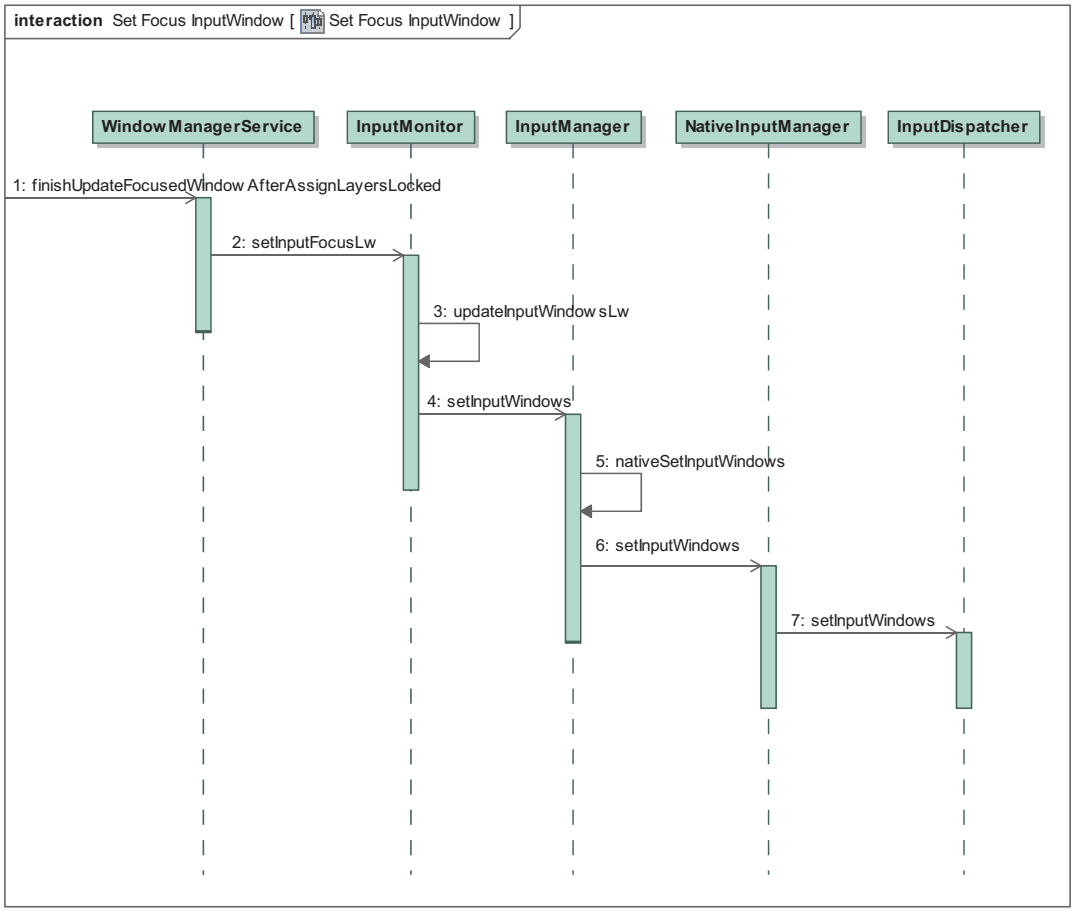


图14-11　系统当前激活的应用程序窗口的注册过程

这个过程一共分为7个步骤，下面我们就详细分析每一个步骤。

Step 1：WindowManagerService.finishUpdateFocusedWindowAfterAssignLayersLocked

frameworks/base/services/java/com/android/server/WindowManagerService.java

```
01 public class WindowManagerService extends IWindowManager.Stub
02         implements Watchdog.Monitor {
03     ......
04
05     private void finishUpdateFocusedWindowAfterAssignLayersLocked() {
06         mInputMonitor.setInputFocusLw(mCurrentFocus);
07     }
08     ......
09
10     InputMonitor mInputMonitor = new InputMonitor();
11     ......
12 }
```

Window管理服务WindowManagerService的成员变量mInputMonitor指向了一个InputMonitor对象，它

是用来辅助管理系统的输入事件的，第6行代码调用它的成员函数setInputFocusLw将系统当前激活的应用程序窗口设置到它里面去。

注意

Window管理服务WindowManagerService的成员变量mCurrentFocus所指向的一个WindowState对象就是用来描述系统当前激活的应用程序窗口的，在前面的14.3.2小节中，我们刚刚将与它所关联的一个Server端InputChannel注册到了C++层的InputDispatcher中。

Step 2：InputMonitor. setInputFocusLw

frameworks/base/services/java/com/android/server/WindowManagerService.java

```
01 public class WindowManagerService extends IWindowManager.Stub
02         implements Watchdog.Monitor {
03     ......
04
05     final class InputMonitor {
06         // Current window with input focus for keys and other non-touch events.  May be null.
07         private WindowState mInputFocus;
08         ......
09
10         public void setInputFocusLw(WindowState newWindow) {
11             ......
12
13             if (newWindow != mInputFocus) {
14                 ......
15
16                 mInputFocus = newWindow;
17                 updateInputWindowsLw();
18             }
19         }
20
21         ......
22     }
23
24     ......
25 }
```

InputMonitor类的成员变量mInputFocus用来描述系统当前激活的应用程序窗口。由于现在系统当前激活的应用程序窗口已经发生了变化，即系统当前激活的应用程序窗口已经更新为参数newWindow所描述的一个应用程序窗口，因此，接下来第16行代码就会将它保存在InputMonitor类的成员变量mInputFocus中，并且第17行代码会调用InputMonitor类的成员函数updateInputWindowsLw将它注册到InputDispatcher中。

Step 3：InputMonitor.updateInputWindowsLw

frameworks/base/services/java/com/android/server/WindowManagerService.java

```
01 public class WindowManagerService extends IWindowManager.Stub
02         implements Watchdog.Monitor {
03     ......
04
05     final ArrayList<WindowState> mWindows = new ArrayList<WindowState>();
06     ......
07
08     final class InputMonitor {
09         ......
10
11         // Temporary list of windows information to provide to the input dispatcher.
12         private InputWindowList mTempInputWindows = new InputWindowList();
```

```
13          ......
14
15          public void updateInputWindowsLw() {
16              ......
17
18              final ArrayList<WindowState> windows = mWindows;
19              final int N = windows.size();
20              for (int i = N - 1; i >= 0; i--) {
21                  final WindowState child = windows.get(i);
22                  if (child.mInputChannel == null || child.mRemoved) {
23                      // Skip this window because it cannot possibly receive input.
24                      continue;
25                  }
26                  ......
27
28                  final boolean hasFocus = (child == mInputFocus);
29                  ......
30
31                  // Add a window to our list of input windows.
32                  final InputWindow inputWindow = mTempInputWindows.add();
33                  inputWindow.inputChannel = child.mInputChannel;
34                  ......
35                  inputWindow.hasFocus = hasFocus;
36                  ......
37              }
38
39              // Send windows to native code.
40              mInputManager.setInputWindows(mTempInputWindows.toNullTerminatedArray());
41
42              .....
43              mTempInputWindows.clear();.
44          }
45
46          ......
47      }
48
49      ......
50  }
```

在前面14.3.1小节的Step 3中提到，Window管理服务WindowManagerService的成员变量mWindows保存了那些需要接收键盘事件的应用程序窗口，第20行到第37行的for循环分别将它们封装成一个InputWindow对象，以便接下来可以重新将它们注册到InputDispatcher中。

注意

如果一个应用程序窗口已经不再需要接收键盘事件了，或者已经被移除了，即第22行的if语句为true，那么它就不会被重新注册到InputDispatcher中，这样就可以保证它不会再接收到键盘事件了。

另外，只有用来封装系统当前激活的应用程序窗口的InputWindow对象的成员变量hasFocus的值才会被设置为true，如第28行和第35行代码所示，这样InputDispatcher就可以知道系统当前激活的应用程序窗口是哪一个了。

执行完成第20行到第37行的for循环之后，需要接收键盘事件的应用程序窗口就保存在一个临时列表mTempInputWindows中了。接下来第40行代码就会调用InputManager类的成员函数setInputWindows将它们重新注册到InputDispatcher中，以便InputDispatcher可以知道哪些应用程序窗口是需要接收键盘事件的，以及哪一个应用程序窗口是系统当前激活的应用程序窗口。

Step 4：InputManager.setInputWindows

frameworks/base/services/java/com/android/server/InputManager.java

```
01 public class InputManager {
02     ......
03
04     private static native void nativeSetInputWindows(InputWindow[] windows);
05     ......
06
07     public void setInputWindows(InputWindow[] windows) {
08         nativeSetInputWindows(windows);
09     }
10
11     ......
12 }
```

第8行代码调用了InputManager类的成员函数nativeSetInputWindows将那些需要接收键盘事件的应用程序窗口注册到InputDispatcher中。

Step 5：InputManager.nativeSetInputWindows

frameworks/base/services/jni/com_android_server_InputManager.cpp

```
1 static void android_server_InputManager_nativeSetInputWindows(JNIEnv* env, jclass clazz,
2         jobjectArray windowObjArray) {
3     ......
4
5     gNativeInputManager->setInputWindows(env, windowObjArray);
6 }
```

InputManager类的成员函数nativeSetInputWindows是一个JNI方法，它是由C++层的函数android_server_InputManager_nativeSetInputWindows来实现的。

在前面14.2.1小节的Step 3中提到，全局变量gNativeInputManager指向了一个C++层的NativeInputManager对象，第5行代码调用它的成员函数setInputWindows来继续将那些需要接收键盘事件的应用程序窗口注册到InputDispatcher中。

Step 6：NativeInputManager.setInputWindows

frameworks/base/services/jni/com_android_server_InputManager.cpp

```
01 void NativeInputManager::setInputWindows(JNIEnv* env, jobjectArray windowObjArray) {
02     Vector<InputWindow> windows;
03
04     jsize length = env->GetArrayLength(windowObjArray);
05     for (jsize i = 0; i < length; i++) {
06         jobject inputTargetObj = env->GetObjectArrayElement(windowObjArray, i);
07         if (! inputTargetObj) {
08             break; // found null element indicating end of used portion of the array
09         }
10
11         windows.push();
12         InputWindow& window = windows.editTop();
13         bool valid = populateWindow(env, inputTargetObj, window);
14         if (! valid) {
15             windows.pop();
16         }
17
18         env->DeleteLocalRef(inputTargetObj);
19     }
20
21     mInputManager->getDispatcher()->setInputWindows(windows);
22 }
```

参数windowObjArray指向了一个Java层的InputWindow对象列表，用来描述那些需要接收键盘事件的应用程序窗口，第5行到第19行的for循环分别调用函数populateWindow将它们转换为C++层的InputWindow对象，并且保存在向量windows中。

在前面14.2.1小节的Step 3中提到，NativeInputManager类的成员变量mInputManager指向了一个C++层的InputManager对象，第21行代码首先获得它内部的一个InputDispatcher对象，接着再调用这个InputDispatcher对象的成员函数setInputWindows来注册那些需要接收键盘事件的应用程序窗口。

Step 7：InputDispatcher.setInputWindows

frameworks/base/libs/ui/InputDispatcher.cpp

```
01 void InputDispatcher::setInputWindows(const Vector<InputWindow>& inputWindows) {
02     ......
03
04     { // acquire lock
05         AutoMutex _l(mLock);
06
07         // Clear old window pointers.
08         ......
09         if (mFocusedWindow) {
10             ......
11             mFocusedWindow = NULL;
12         }
13
14         mWindows.clear();
15
16         // Loop over new windows and rebuild the necessary window pointers for
17         // tracking focus and touch.
18         mWindows.appendVector(inputWindows);
19
20         size_t numWindows = mWindows.size();
21         for (size_t i = 0; i < numWindows; i++) {
22             const InputWindow* window = & mWindows.itemAt(i);
23             if (window->hasFocus) {
24                 mFocusedWindow = window;
25                 break;
26             }
27         }
28
29         ......
30     } // release lock
31
32     ......
33 }
```

在InputDispatcher类中，成员变量mWindows用来保存系统中所有需要接收键盘事件的应用程序窗口，而成员变量mFocusedWindow用来保存系统当前激活的应用程序窗口。第9行到第14行代码首先将它们清空，因为接下来需要重新注册系统中需要接收键盘事件的应用程序窗口。

由于保存在参数inputWindows中的应用程序窗口即为系统当前需要接收键盘事件的应用程序窗口，因此，第18行代码首先将它们保存在InputDispatcher类的成员变量mWindows中，接着第21行到第27行的for循环再从它们之中找到系统当前激活的应用程序窗口，并且保存在InputDispatcher类的成员变量mFocusedWindow中。

这一步执行完成之后，系统当前激活的应用程序窗口就注册到InputDispatcher中了。返回到前面14.3.1小节的Step 1中，即ViewRoot类的成员函数setView中，接下来我们继续分析Client端InputChannel的注册过程。

14.3.4 注册Client端InputChannel

从前面14.3.1小节的Step 1可以知道，InputQueue类的静态成员函数registerInputChannel是用来注册一个Client端InputChannel的，接下来我们就从这个函数开始分析一个Client端InputChannel的注册过程，如图14-12所示。

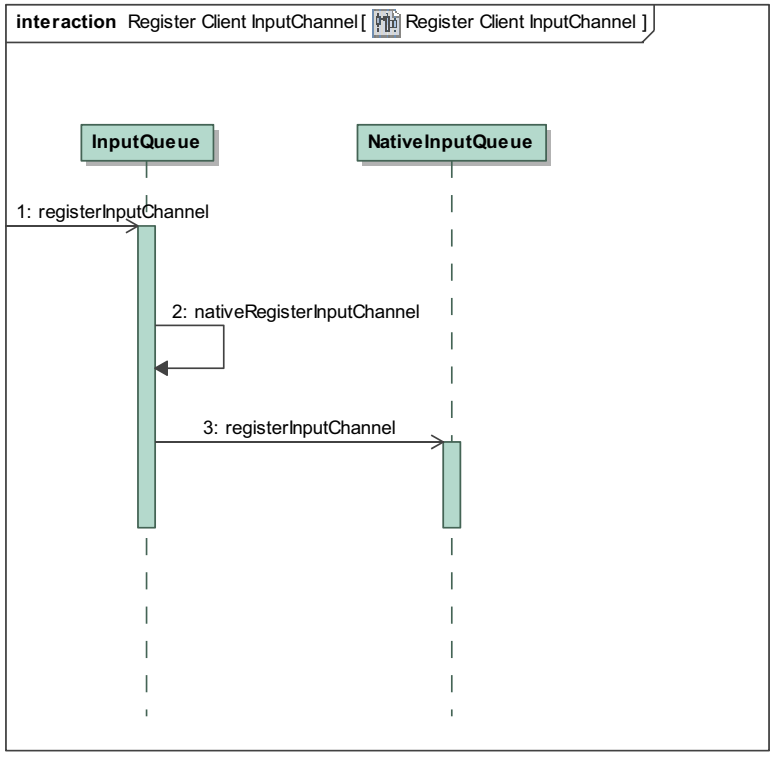


图14-12 Client端InputChannel的注册过程

这个过程一共分为3个步骤，下面我们就详细分析每一个步骤。

Step 1：InputQueue.registerInputChannel

frameworks/base/core/java/android/view/InputQueue.java

```
01 public final class InputQueue {
02     ......
03
04     private static native void nativeRegisterInputChannel(InputChannel inputChannel,
05             InputHandler inputHandler, MessageQueue messageQueue);
06     ......
07
08     public static void registerInputChannel(InputChannel inputChannel, InputHandler inputHandler,
09             MessageQueue messageQueue) {
10         ......
11
12         synchronized (sLock) {
13             ......
14
15             nativeRegisterInputChannel(inputChannel, inputHandler, messageQueue);
16         }
17     }
18
19     ......
20 }
```

注意

InputQueue类的静态成员函数registerInputChannel是在系统当前激活的应用程序窗口所运行在的应用程序进程中执行的，它有三个参数inputChannel、inputHandler和messageQueue。其中，参数inputChannel用来描述一个Client端InputChannel，它是在前面的14.3.1小节中创建的；参数inputHandler指向了ViewRoot类中的成员变量mInputHandler所描述的一个InputHandler对象，它是用来接收InputDispatcher分发过来的键盘事件的；参数messageQueue指向了系统当前激活的应用程序窗口所运行在的应用程序进程的主线程的一个消息队列。

第15行代码调用InputQueue类的静态成员函数nativeRegisterInputChannel来注册一个Client端InputChannel。

Step 2：InputQueue.nativeRegisterInputChannel

frameworks/base/core/jni/android_view_InputQueue.cpp

```
01 static NativeInputQueue gNativeInputQueue;
02 ......
03
04 static void android_view_InputQueue_nativeRegisterInputChannel(JNIEnv* env, jclass clazz,
05         jobject inputChannelObj, jobject inputHandlerObj, jobject messageQueueObj) {
06     status_t status = gNativeInputQueue.registerInputChannel(
07        env, inputChannelObj, inputHandlerObj, messageQueueObj);
08
09     ......
10 }
```

InputQueue类的静态成员函数nativeRegisterInputChannel是一个JNI方法，它是由C++层的函数android_view_InputQueue_nativeRegisterInputChannel来实现的。

全局变量gNativeInputQueue指向的是一个C++层的NativeInputQueue对象，第6行代码调用它的成员函数registerInputChannel将参数inputChannelObj所描述的一个Client端InputChannel注册到系统当前激活的应用程序窗口所运行在的应用程序进程的主线程中。

Step 3：NativeInputQueue.registerInputChannel

frameworks/base/core/jni/android_view_InputQueue.cpp

```
01 status_t NativeInputQueue::registerInputChannel(JNIEnv* env, jobject inputChannelObj,
02         jobject inputHandlerObj, jobject messageQueueObj) {
03     sp<InputChannel> inputChannel = android_view_InputChannel_getInputChannel(env,
04         inputChannelObj);
05     ......
06
07     sp<Looper> looper = android_os_MessageQueue_getLooper(env, messageQueueObj);
08
09     { // acquire lock
10         AutoMutex _l(mLock);
11
12         if (getConnectionIndex(inputChannel) >= 0) {
13             ......
14             return BAD_VALUE;
15         }
16
17         uint16_t connectionId = mNextConnectionId++;
18         sp<Connection> connection = new Connection(connectionId, inputChannel, looper);
19         status_t result = connection->inputConsumer.initialize();
20         ......
21
22         connection->inputHandlerObjGlobal = env->NewGlobalRef(inputHandlerObj);
```

```
23
24          int32_t receiveFd = inputChannel->getReceivePipeFd();
25          mConnectionsByReceiveFd.add(receiveFd, connection);
26
27          looper->addFd(receiveFd, 0, ALOOPER_EVENT_INPUT, handleReceiveCallback, this);
28      } // release lock
29
30      ......
31      return OK;
32  }
```

参数inputChannelObj指向了一个Java层的Client端InputChannel，第3行和第4行代码首先调用函数android_view_InputChannel_getInputChannel将它转换为一个C++层的Client端InputChannel，并且保存在变量inputChannel中。

第7行代码调用函数android_os_MessageQueue_getLooper来获得系统当前激活的应用程序窗口所运行在的应用程序进程的主线程的一个Looper对象，并且将它保存在变量looper中。

第12行的if语句调用NativeInputQueue类的成员函数getConnectionIndex来检查变量inputChannel所描述的一个Client端InputChannel是否已经注册过了。如果已经注册过，那么第14行代码就直接出错返回了。

注册到一个应用程序进程的主线程中的Client端InputChannel首先会被封装成一个Connection对象，然后再以这个Client端InputChannel内部的一个前向管道的读端文件描述符为关键字，保存在NativeInputQueue类的成员变量mConnectionsByReceiveFd中。

第18行代码首先将变量inputChannel所描述的一个Client端InputChannel封装成一个Connection对象。每一个Connection对象内部都有一个InputConsumer对象，它是用来接收InputDispatcher分发过来的键盘事件的。接着第19行代码用来初始化前面所创建的一个Connection对象的内部的一个InputConsumer对象。

InputConsumer类的成员函数initialize的实现如下所示。

frameworks/base/libs/ui/InputTransport.cpp

```
01 status_t InputConsumer::initialize() {
02      ......
03
04      int ashmemFd = mChannel->getAshmemFd();
05      int result = ashmem_get_size_region(ashmemFd);
06      ......
07
08      mAshmemSize = (size_t) result;
09
10      mSharedMessage = static_cast<InputMessage*>(mmap(NULL, mAshmemSize,
11              PROT_READ | PROT_WRITE, MAP_SHARED, ashmemFd, 0));
12      ......
13
14      return OK;
15  }
```

InputConsumer类的成员变量mChannel指向了一个C++层的Client端InputChannel，第4行代码首先调用它的成员函数getAshmemFd来获得它所引用的一块匿名共享内存的一个文件描述符ashmemFd；接着第5行代码再以文件描述符ashmemFd为参数来调用运行时库cutils所提供的一个匿名共享内存访问接口ashmem_get_size_region，以便可以获得它所描述的一块匿名共享内存的大小；最后第8行代码将这块匿名共享内存的大小保存在InputConsumer类的成员变量mAshmemSize中。

第10行和第11行代码首先调用函数mmap将文件描述符ashmemFd所描述的一块匿名共享内存映射到当前进程的地址空间，即映射到系统当前激活的应用程序窗口所运行在的应用程序进程的地址空间；接着再将映射后所得到的地址空间内容结构化成一个InputMessage结构体，并且保存在InputConsumer类的成员变量mSharedMessage中。在接下来的14.4小节中，我们就会看到，这个InputMessage结构体是用

来描述InputDispatcher分发过来的一个键盘事件数据的。

回到NativeInputQueue类的成员函数registerInputChannel中，接下来第22行代码首先将参数inputHandlerObj所描述的一个InputHandler对象保存在前面所创建的一个Connection对象的成员变量inputHandlerObjGlobal中；接着第24行代码再获得变量inputChannel所描述的一个Client端InputChannel的内部的一个前向管道的读端文件描述符；最后第25行代码就以这个文件描述符为关键字，将前面所创建的一个Connection对象保存在NativeInputQueue类的成员变量mConnectionsByReceiveFd中。

第27行代码调用前面所获得的一个Looper对象的成员函数addFd将前面所注册的一个Client端InputChannel的内部的一个前向管道的读端文件描述符保存在它里面，以便可以监控这个前向管道的IO写事件。当这个前向管道发生了一个IO写事件时，即当这个前向管道被写入了一个新的数据时，NativeInputQueue类的静态成员函数handleReceiveCallback就会被调用。

注意

如果这时候这个Looper对象所运行在的应用程序进程的主线程正处于睡眠等待状态，那么它就会首先被唤醒，然后再调用NativeInputQueue类的静态成员函数handleReceiveCallback。

在接下来的14.4小节中，我们就会看到，InputDispatcher在分发一个键盘事件给系统当前激活的应用程序窗口处理时，会向与系统当前激活的应用程序窗口所关联的一个Server端InputChannel的内部的一个前向管道写入一个新的数据，将系统当前激活的应用程序窗口所运行在的应用程序进程的主线程唤醒，以便它可以接收InputDispatcher分发过来的一个键盘事件。

这一步执行完成之后，一个Client端InputChannel就注册完成了。至此，系统当前激活的应用程序窗口就与输入管理器InputManager建立好一个连接了，以后它就可以接收到系统的键盘事件了。

14.4　键盘消息的分发过程

从前面14.2.3和14.2.4小节的内容可以知道，当系统没有键盘事件发生时，InputReader和InputDispatcher就会处于睡眠等待状态。从前面14.3.4小节的内容又可以知道，这时候如果系统当前激活的应用程序窗口所运行在的应用程序进程的主线程无事可做，那么它就会在消息循环中等待InputDispatcher分发键盘事件给它处理。

一旦系统发生键盘事件， InputReader首先就会被唤醒，并且获得刚才所发生的键盘事件；接着InputReader再将InputDispatcher唤醒，以便它可以将刚才所发生的键盘事件分发给系统当前激活的应用程序窗口处理。这时候InputDispatcher又会再次进入睡眠等待状态，直到刚才所发生的键盘事件被系统当前激活的应用程序窗口处理完成为止。

系统当前激活的应用程序窗口并不是直接接收到InputDispatcher分发过来的键盘事件的。首先是系统当前激活的应用程序窗口所运行在的应用程序进程的主线程接收到InputDispatcher分发过来的键盘事件，然后这个线程再将接收到的键盘事件封装成一个键盘消息发送到自己的消息队列中，最后这个键盘消息会被分发给系统当前激活的应用程序窗口处理。

系统当前激活的应用程序窗口在处理完成一个键盘消息之后，就会将InputDispatcher唤醒，以便它可以知道刚才所发生的键盘事件已经被处理完成，因此，它就可以开始新一轮的键盘事件分发工作了。

接下来，我们首先分析InputReader获得键盘事件的过程，然后分析InputDispatcher分发键盘事件的过程，以及系统当前激活的应用程序窗口获得键盘消息的过程，最后分析InputDispatcher获得键盘事件处理完成通知的过程。

14.4.1 InputReader获得键盘事件

从前面14.2.4小节的Step 2可以知道，当系统发生了一个新的键盘事件时，InputReader就会被唤醒，并且调用它的成员函数process来对新发生的键盘事件进行处理，如图14-13所示。

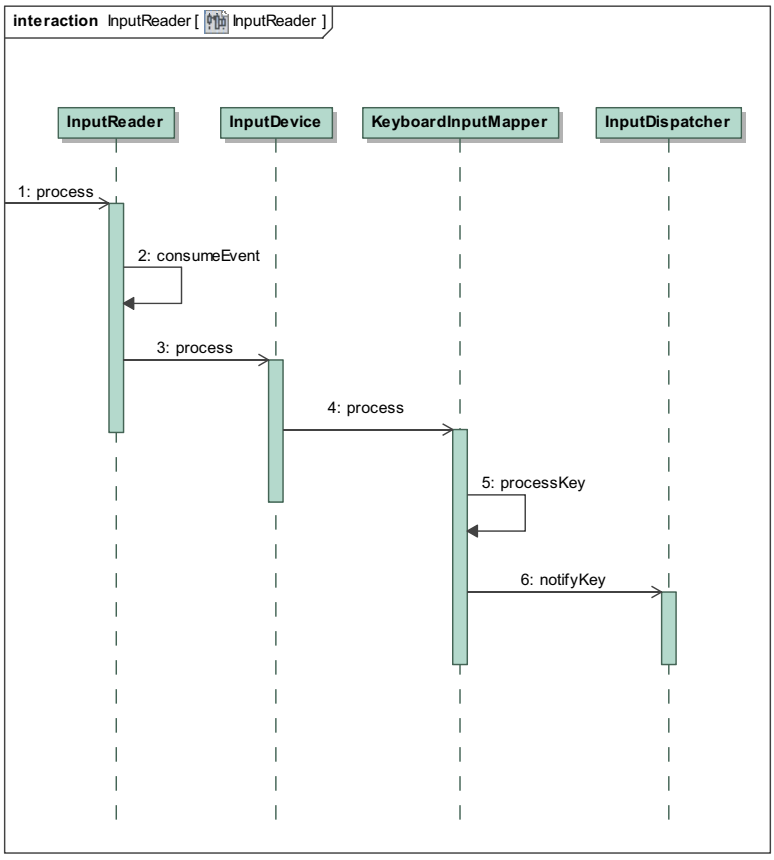


图14-13 InputReader获得键盘事件的过程

这个过程一共分为6个步骤，下面我们就详细分析每一个步骤。

Step 1：InputReader.process

frameworks/base/libs/ui/InputReader.cpp

```
01 void InputReader::process(const RawEvent* rawEvent) {
02     switch (rawEvent->type) {
03     case EventHubInterface::DEVICE_ADDED:
04         ......
05         break;
06
07     case EventHubInterface::DEVICE_REMOVED:
08         ......
09         break;
10
11     case EventHubInterface::FINISHED_DEVICE_SCAN:
12         ......
13         break;
14
15     default:
16         consumeEvent(rawEvent);
17         break;
18     }
19 }
```

参数rawEvent用来描述一个刚刚发生的输入事件。在我们这个场景中，假设刚刚发生的输入事件是一个键盘事件，即参数rawEvent的成员变量type的值等于EV_KEY。接下来第16行代码就会调用InputReader类的成员函数consumeEvent来处理刚刚发生的一个键盘事件。

Step 2：InputReader.consumeEvent

frameworks/base/libs/ui/InputReader.cpp

```
01 void InputReader::consumeEvent(const RawEvent* rawEvent) {
02     int32_t deviceId = rawEvent->deviceId;
03
04     { // acquire device registry reader lock
05         RWLock::AutoRLock _rl(mDeviceRegistryLock);
06
07         ssize_t deviceIndex = mDevices.indexOfKey(deviceId);
08         if (deviceIndex < 0) {
09             LOGW("Discarding event for unknown deviceId %d.", deviceId);
10             return;
11         }
12
13         InputDevice* device = mDevices.valueAt(deviceIndex);
14         if (device->isIgnored()) {
15             //LOGD("Discarding event for ignored deviceId %d.", deviceId);
16             return;
17         }
18
19         device->process(rawEvent);
20     } // release device registry reader lock
21 }
```

在InputReader类中，每一个被监控的输入设备都使用一个InputDevice对象来描述。这些InputDevice对象以它们所描述的输入设备的ID值为关键字，保存在InputReader类的成员变量mDevices中。

第2行代码首先获得刚刚发生了一个键盘事件的输入设备的ID值，接着第7行到第13行代码以这个ID值为关键字在InputReader类的成员变量mDevices中找到一个对应的InputDevice对象，并且将它保存在变量device中。

第14行的if语句检查是否需要忽略InputDevice对象device所描述的输入设备的IO输入事件。如果需要忽略，那么接下来第16行代码就直接返回了；否则，第19行代码就会调用InputDevice对象device的成员函数process来进一步处理参数 rawEvent所描述的一个键盘事件。

Step 3：InputDevice.process

frameworks/base/libs/ui/InputReader.cpp

```
1 void InputDevice::process(const RawEvent* rawEvent) {
2     size_t numMappers = mMappers.size();
3     for (size_t i = 0; i < numMappers; i++) {
4         InputMapper* mapper = mMappers[i];
5         mapper->process(rawEvent);
6     }
7 }
```

在Android系统中，一个输入设备的类型可能是复合的，即它可以产生多种类型的IO输入事件。例如，一个输入设备可能既包含键盘按钮，又包含轨迹球，这样它就既可以产生键盘输入事件，又可以产生轨迹球输入事件。

在InputDevice类中，每一种类型的IO输入事件都是由一个专门的InputMapper对象来处理的，这些InputMapper对象保存在InputDevice类的成员变量mMappers中。例如，键盘输入事件由一个类型为KeyboardKeyMapper的InputMapper对象来处理，而轨迹球输入事件由一个类型为TrackballInputMapper的InputMapper对象来处理。

注意

虽然第3行到第6行的for循环会调用保存在InputDevice类的成员变量mMappers中的每一个InputMapper对象来处理参数rawEvent所描述的一个输入事件，但是由于这些InputMapper对象会在内部忽略与自己无关的输入事件，因此，最终只会有一个InputMapper对象对参数rawEvent所描述的一个输入事件进行处理。

在我们这个场景中，由于参数rawEvent所描述的是一个键盘事件，即它最终是由一个类型为KeyboardKeyMapper的InputMapper对象来处理的，因此，接下来我们就继续分析KeyboardKeyMapper类的成员函数process的实现。

Step 4：KeyboardInputMapper.process

frameworks/base/libs/ui/InputReader.cpp

```
01 void KeyboardInputMapper::process(const RawEvent* rawEvent) {
02     switch (rawEvent->type) {
03     case EV_KEY: {
04         int32_t scanCode = rawEvent->scanCode;
05         if (isKeyboardOrGamepadKey(scanCode)) {
06             processKey(rawEvent->when, rawEvent->value != 0, rawEvent->keyCode, scanCode,
07                     rawEvent->flags);
08         }
09         break;
10     }
11     }
12 }
```

从这里就可以看出，KeyboardInputMapper类只对类型为EV_KEY的IO输入事件进行处理，即只对键盘事件进行处理。

第4行代码首先获得参数rawEvent所描述的一个键盘事件的扫描码，接着第5行的if语句检查这个扫描码是否对应于一个合法的键盘按键。如果是，那么接下来第6行和第7行代码就会调用KeyboardInputMapper类的成员函数processKey来继续处理参数rawEvent所描述的一个键盘事件。

Step 5：KeyboardInputMapper.processKey

frameworks/base/libs/ui/InputReader.cpp

```
01 void KeyboardInputMapper::processKey(nsecs_t when, bool down, int32_t keyCode,
02         int32_t scanCode, uint32_t policyFlags) {
03     int32_t newMetaState;
04     nsecs_t downTime;
05     bool metaStateChanged = false;
06
07     { // acquire lock
08      AutoMutex _l(mLock);
09
10      if (down) {
11          // Rotate key codes according to orientation if needed.
12          // Note: getDisplayInfo is non-reentrant so we can continue holding the lock.
13          if (mAssociatedDisplayId >= 0) {
14              int32_t orientation;
15              if (! getPolicy()->getDisplayInfo(mAssociatedDisplayId, NULL, NULL, & orientation)) {
16                  return;
17              }
18
19              keyCode = rotateKeyCode(keyCode, orientation);
20          }
21
22          // Add key down.
23          ssize_t keyDownIndex = findKeyDownLocked(scanCode);
```

```
24          if (keyDownIndex >= 0) {
25              // key repeat, be sure to use same keycode as before in case of rotation
26              keyCode = mLocked.keyDowns.itemAt(keyDownIndex).keyCode;
27          } else {
28              ......
29
30              mLocked.keyDowns.push();
31              KeyDown& keyDown = mLocked.keyDowns.editTop();
32              keyDown.keyCode = keyCode;
33              keyDown.scanCode = scanCode;
34          }
35
36          mLocked.downTime = when;
37      } else {
38          // Remove key down.
39          ssize_t keyDownIndex = findKeyDownLocked(scanCode);
40          if (keyDownIndex >= 0) {
41              // key up, be sure to use same keycode as before in case of rotation
42              keyCode = mLocked.keyDowns.itemAt(keyDownIndex).keyCode;
43              mLocked.keyDowns.removeAt(size_t(keyDownIndex));
44          } else {
45              ......
46              return;
47          }
48      }
49
50      int32_t oldMetaState = mLocked.metaState;
51      newMetaState = updateMetaState(keyCode, down, oldMetaState);
52      if (oldMetaState != newMetaState) {
53          mLocked.metaState = newMetaState;
54          ......
55      }
56
57      downTime = mLocked.downTime;
58     } // release lock
59     ......
60
61     getDispatcher()->notifyKey(when, getDeviceId(), AINPUT_SOURCE_KEYBOARD, policyFlags,
62         down ? AKEY_EVENT_ACTION_DOWN : AKEY_EVENT_ACTION_UP,
63         AKEY_EVENT_FLAG_FROM_SYSTEM, keyCode, scanCode, newMetaState, downTime);
64 }
```

当系统发生一个键盘事件时，可能是一个键盘按键被按下了，也有可能是一个键盘按键被松开了，这是通过参数down来描述的。此外，当一个键盘按键被按下或者松开时，可能有其他的键盘按键正处于被按下状态，这些处于被按下状态的其他键盘按键保存在KeyboardInputMapper类的成员变量mLocked中。

第13行到第36行代码用来处理键盘按键被按下的情况。

一个键盘事件所对应的键盘码除了与键盘布局有关之外，还与当前的屏幕方向有关。例如，当用户按下的是DPAD的一个方向键时，根据当时屏幕方向的不同，这个方向键对应的键盘码是不一样的，因此，我们就需要对它进行调整。

参数scanCode和keyCode保存的分别是当前所发生的键盘事件所对应的扫描码和键盘码，其中，键盘码是根据键盘布局转换扫描码得到的，具体可以参考前面14.2.4小节中Step 3的内容。第13行到第20行代码根据当前的屏幕方向来进一步调整当前所发生的键盘事件的键盘码。

KeyboardInputMapper类的成员变量mAssociatedDisplayId保存的是当前屏幕的ID值。Android系统的设计是支持多屏显示的，但是目前只支持一个，因此，KeyboardInputMapper类的成员变量mAssociatedDisplayId的值就被初始化为0。

第15行代码首先调用父类InputMapper的成员函数getPolicy来获得一个InputReaderPolicyInterface接口；

然后调用这个InputReaderPolicyInterface接口的成员函数getDisplayInfo来获得ID值为mAssociatedDisplayId的屏幕的方向，即它是横着的还是竖着的；最后第19行代码再根据前面所得到的屏幕方向来调整当前所发生的键盘事件的键盘码，这是通过调用函数rotateKeyCode来实现的。

注意

调用InputMapper类的成员函数getPolicy所获得的InputReaderPolicyInterface接口指向了在前面14.2.1小节的Step 3中所创建的一个NativeInputManager对象。

第23行代码调用KeyboardInputMapper类的成员函数findKeyDownLocked来检查当前被按下的键盘按键是否是被重复按下的。如果是，那么接下来第26行代码就需要将它的键盘码恢复为上一次被按下时的键盘码；否则，第30行到第33行代码就需要将它封装成一个KeyDown结构体，并且保存在KeyboardInputMapper类的成员变量mLocked中。

注意

当一个键盘按键被重复按下时，它的键盘码不应该受到屏幕方向的影响；否则，就会影响用户体验。

第39行到第47行代码用来处理键盘按键被松开的情况。

第39行代码首先调用KeyboardInputMapper类的成员函数findKeyDownLocked来检查当前被松开的键盘按键是否是上次按下的。如果是，那么接下来第42行和第43行代码就会将它从KeyboardInputMapper类的成员变量mLocked中移除；否则，就说明系统内部发生了错误。因此，接下来第46行代码就直接返回了，即不再继续对当前所发生的键盘事件进行处理了。

第50行到第55行代码用来更新系统当前的组合键状态，即Alt键和Shift键的按下状态。

系统当前的组合键状态记录在mLocked的成员变量metaState中，第50行代码首先将它的值取出来，并且保存在变量oldMetaState中；接着第51行代码调用函数updateMetaState来得到系统新的组合键状态，并且保存在变量newMetaState中。如果变量oldMetaState的值不等于变量newMetaState的值，即第52行的if语句为true，那么就说明当前被按下或者被松开的是一个Alt键或者一个Shift键。因此，接下来第53行代码就会将变量newMetaState的值设置到mLocked的成员变量metaState中，以便可以正确地反映系统当前的组合键状态。

最后，第61行到第63行代码首先调用父类InputMapper的成员函数getDispatcher来获得在前面14.2.1小节的Step 3中所创建的一个InputDispatcher对象；接着再调用这个InputDispatcher对象的成员函数notifyKey来通知它系统刚刚发生了一个键盘事件，以便它可以将这个键盘事件分发给系统当前激活的应用程序窗口处理。

接下来，我们就继续分析InputDispatcher类的成员函数notifyKey的实现。

Step 6：InputDispatcher.notifyKey

frameworks/base/libs/ui/InputDispatcher.cpp

```
01 void InputDispatcher::notifyKey(nsecs_t eventTime, int32_t deviceId, int32_t source,
02     uint32_t policyFlags, int32_t action, int32_t flags,
03     int32_t keyCode, int32_t scanCode, int32_t metaState, nsecs_t downTime) {
04     ......
05
06     if (! validateKeyEvent(action)) {
07         return;
08     }
09     ......
10
11     if (policyFlags & POLICY_FLAG_ALT) {
```

```
12          metaState |= AMETA_ALT_ON | AMETA_ALT_LEFT_ON;
13      }
14      if (policyFlags & POLICY_FLAG_ALT_GR) {
15          metaState |= AMETA_ALT_ON | AMETA_ALT_RIGHT_ON;
16      }
17      if (policyFlags & POLICY_FLAG_SHIFT) {
18          metaState |= AMETA_SHIFT_ON | AMETA_SHIFT_LEFT_ON;
19      }
20
21      policyFlags |= POLICY_FLAG_TRUSTED;
22      mPolicy->interceptKeyBeforeQueueing(eventTime, deviceId, action, /*byref*/ flags,
23          keyCode, scanCode, /*byref*/ policyFlags);
24
25      bool needWake;
26      { // acquire lock
27          AutoMutex _l(mLock);
28
29          int32_t repeatCount = 0;
30          KeyEntry* newEntry = mAllocator.obtainKeyEntry(eventTime,
31              deviceId, source, policyFlags, action, flags, keyCode, scanCode,
32              metaState, repeatCount, downTime);
33
34          needWake = enqueueInboundEventLocked(newEntry);
35      } // release lock
36
37      if (needWake) {
38          mLooper->wake();
39      }
40  }
```

参数action用来描述一个键盘按键是被按下了还是被松开了，它的值要么等于AKEY_EVENT_ACTION_DOWN，要么等于AKEY_EVENT_ACTION_UP。第6行代码首先调用函数validateKeyEvent来验证它的合法性。如果不合法，那么接下来第7行代码就直接返回了，即不再对当前所发生的键盘事件进行处理了。

参数policyFlags是用来控制键盘事件的分发策略的，第11行、第14行和第17行的三个if语句分别检查它的第POLICY_FLAG_ALT位、第POLICY_FLAG_ALT_GR位和第POLICY_FLAG_SHIFT位的值是否等于1。如果它们分别等于1，那么第12行、第15行和第18行代码就会认为系统的左Alt键、右Alt键和左Shift键处于被按下状态。

InputDispatcher类有一个类型为InputDispatcherPolicyInterface的成员变量mPolicy，它指向了在前面14.2.1小节的Step 3中所创建的一个NativeInputManager对象，第22行和第23行代码调用它的成员函数interceptKeyBeforeQueueing来截获系统当前所发生的键盘事件。NativeInputManager类的成员函数interceptKeyBeforeQueueing最终会将截获到的键盘事件交给Window管理服务WindowManagerService和Power管理服务PowerManagerService来处理，以便它们可以设置屏幕的亮度，以及管理设备的电源状态等。

InputDispatcher类的成员变量mAllocator指向了一个Allocator对象，第30行到第32行代码首先调用它的成员函数obtainKeyEntry将当前所发生的键盘事件封装成一个KeyEntry结构体，接着第34行代码再调用InputDispatcher类的成员函数enqueueInboundEventLocked将这个KeyEntry结构体添加到一个待分发键盘事件队列中。

InputDispatcher类的成员函数enqueueInboundEventLocked的实现如下所示。

frameworks/base/libs/ui/InputDispatcher.cpp

```
01 bool InputDispatcher::enqueueInboundEventLocked(EventEntry* entry) {
02     bool needWake = mInboundQueue.isEmpty();
03     mInboundQueue.enqueueAtTail(entry);
04
05     switch (entry->type) {
06     case EventEntry::TYPE_KEY: {
```

```
07          KeyEntry* keyEntry = static_cast<KeyEntry*>(entry);
08          if (isAppSwitchKeyEventLocked(keyEntry)) {
09              if (keyEntry->action == AKEY_EVENT_ACTION_DOWN) {
10                  mAppSwitchSawKeyDown = true;
11              } else if (keyEntry->action == AKEY_EVENT_ACTION_UP) {
12                  if (mAppSwitchSawKeyDown) {
13                      ......
14                      mAppSwitchDueTime = keyEntry->eventTime + APP_SWITCH_TIMEOUT;
15                      mAppSwitchSawKeyDown = false;
16                      needWake = true;
17                  }
18              }
19          }
20          break;
21      }
22      }
23
24      return needWake;
25  }
```

InputDispatcher类的成员变量mInboundQueue用来描述一个待分发键盘事件队列，所有保存在它里面的键盘事件都是需要分发给系统当前激活的应用程序窗口处理的。

InputDispatcher类的成员函数enqueueInboundEventLocked除了会将一个新发生的键盘事件添加到它的成员变量mInboundQueue所描述的一个待分发键盘事件队列中之外，还会检查是否需要将InputDispatcher唤醒，以便InputDispatcher可以马上去处理刚刚添加到待分发键盘事件队列中的键盘事件。如果需要将InputDispatcher唤醒，那么InputDispatcher类的成员函数enqueueInboundEventLocked的返回值就会等于true。

在两种情况下，InputDispatcher类的成员函数enqueueInboundEventLocked需要将InputDispatcher唤醒。

第一种情况是在将一个新发生的键盘事件添加到待分发键盘事件队列之前，这个待分发键盘事件队列是一个空队列，这时候就说明InputDispatcher正处于睡眠等待状态。现在既然系统发生了一个新的键盘事件，那么就需要将InputDispatcher唤醒来处理了，即将InputDispatcher类的成员函数enqueueInboundEventLocked的返回值neekWake设置为true，如第2行代码所示。

第二种情况是当前发生的键盘事件是与应用程序窗口切换操作相关的。考虑这样一个场景：系统当前激活的应用程序窗口在处理一个键盘事件的过程中，用户又按下了Home键或者Back键，这时候就意味着需要切换到另外一个应用程序窗口中。但是由于InputDispatcher正在等待系统当前激活的应用程序窗口处理完成上一个键盘事件，因此，它就不能马上去分发这个与应用程序窗口切换相关的键盘事件。如果系统当前激活的应用程序窗口很长时间都不通知InputDispatcher它处理完成上一个键盘事件，那么InputDispatcher就会长时间地处于睡眠等待状态，导致系统不能及时地响应用户切换应用程序窗口的操作。这种体验是非常不好的，因为用户会觉得系统没有响应了。为了解决这个问题，InputDispatcher类的成员函数enqueueInboundEventLocked在将一个键盘事件添加到待分发键盘事件队列中之后，就会调用成员函数isAppSwitchKeyEventLocked来检查这个键盘事件是否是与应用程序窗口切换相关的，如第8行代码所示。如果是，那么就需要在对应的键盘按键松开时，马上将InputDispatcher唤醒，即将InputDispatcher类的成员函数enqueueInboundEventLocked的返回值neekWake设置为true，而不管系统当前激活的应用程序窗口是否已经处理完成上一个键盘事件了，如第9行到第18行代码所示。

注意

第14行代码会将与应用程序窗口切换操作相关的一个键盘事件的最迟分发时间设置在InputDispatcher类的成员变量mAppSwitchDueTime中，在接下来的14.4.2小节中，我们就会看到它的使用方法。

回到InputDispatcher类的成员函数notifyKey中，接下来第37行的if语句判断InputDispatcher类的成员函数enqueueInboundEventLocked的返回值needWake是否等于true。如果等于，那么最后第38行代码就会调用InputDispatcher类的成员变量mLooper所指向的一个Looper对象的成员函数wake将InputDispatcher唤醒。

这一步执行完成之后，InputReader就处理完成一个新发生的键盘事件了，并且将InputDispatcher唤醒，以便它可以将这个新发生的键盘事件分发给系统当前激活的应用程序窗口处理。

接下来，我们继续分析InputDispatcher将一个新发生的键盘事件分发给系统当前激活的应用程序窗口处理的过程。

14.4.2　InputDispatcher分发键盘事件

从前面14.2.3小节的Step 2可以知道，InputDispatcher是在InputDispatcher类的成员函数dispatchOnce中进入睡眠等待状态的。当系统发生了一个新的键盘事件时，InputReader就会将InputDispatcher唤醒；InputDispatcher被唤醒之后，就会调用InputDispatcher类的成员函数dispatchOnceInnerLocked将这个新发生的键盘事件分发给系统当前激活的应用程序窗口处理，如图14–14所示。

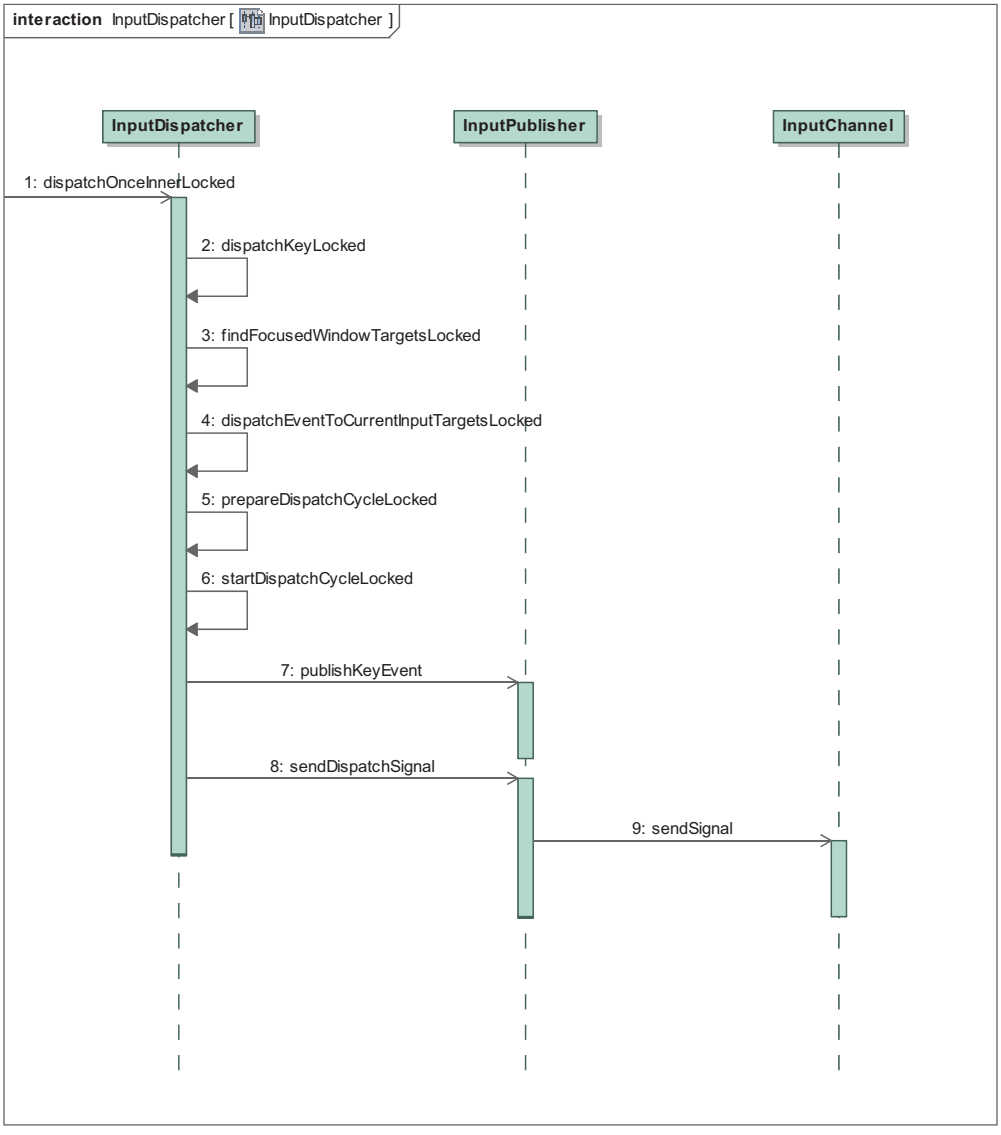


图14–14　InputDispatcher分发键盘事件的过程

这个过程一共分为9个步骤，下面我们就详细分析每一个步骤。

Step 1：InputDispatcher.dispatchOnceInnerLocked

我们分段来阅读InputDispatcher类的成员函数dispatchOnceInnerLocked。

frameworks/base/libs/ui/InputDispatcher.cpp

```
01 void InputDispatcher::dispatchOnceInnerLocked(nsecs_t keyRepeatTimeout,
02     nsecs_t keyRepeatDelay, nsecs_t* nextWakeupTime) {
03     nsecs_t currentTime = now();
04     ......
05
06     bool isAppSwitchDue = mAppSwitchDueTime <= currentTime;
07     if (mAppSwitchDueTime < *nextWakeupTime) {
08         *nextWakeupTime = mAppSwitchDueTime;
09     }
```

从前面14.2.3小节的Step 2可以知道，参数keyRepeatTimeout和keyRepeatDelay是两个与键盘重复按键事件相关的参数；而参数nextWakeupTime是一个输出参数，用来描述InputDispatcher处理下一个键盘事件的最迟时间。

从前面14.4.1小节的Step 6可以知道，当系统发生了一个与应用程序窗口切换操作相关的键盘事件时，InputDispatcher类的成员函数enqueueInboundEventLocked会将这个键盘事件的最迟分发时间保存在InputDispatcher类的成员变量mAppSwitchDueTime中。第6行代码首先判断这个时间是否小于等于系统的当前时间。如果小于等于，那么接下来第8行代码就会将输出参数nextWakeupTime的值设置为这个成员变量的值，以便InputDispatcher处理完成当前需要分发的键盘事件之后，可以马上对这个与应用程序窗口切换操作相关的键盘事件进行处理。

frameworks/base/libs/ui/InputDispatcher.cpp

```
10     // Ready to start a new event.
11     // If we don't already have a pending event, go grab one.
12     if (! mPendingEvent) {
13         if (mInboundQueue.isEmpty()) {
14             ......
15
16             // Synthesize a key repeat if appropriate.
17             if (mKeyRepeatState.lastKeyEntry) {
18                 if (currentTime >= mKeyRepeatState.nextRepeatTime) {
19                     mPendingEvent = synthesizeKeyRepeatLocked(currentTime, keyRepeatDelay);
20                 } else {
21                     if (mKeyRepeatState.nextRepeatTime < *nextWakeupTime) {
22                         *nextWakeupTime = mKeyRepeatState.nextRepeatTime;
23                     }
24                 }
25             }
26             if (! mPendingEvent) {
27                 return;
28             }
29         } else {
30             // Inbound queue has at least one entry.
31             EventEntry* entry = mInboundQueue.headSentinel.next;
32             ......
33
34             mInboundQueue.dequeue(entry);
35             mPendingEvent = entry;
36         }
37
38         ......
39     }
```

InputDispatcher类的成员变量mPendingEvent指向了上一个需要分发给系统当前激活的应用程序窗口处理的键盘事件。这个键盘事件在成功地分发出去之前，会一直保存在InputDispatcher类的成员变量mPendingEvent中。

第12行的if语句检查上一个需要分发给系统当前激活的应用程序窗口处理的键盘事件是否已经成功地分发出去了。如果已经成功地分发出去了，那么接下来第13行到第36行代码就会继续将下一个需要分发的键盘事件取出来，以便可以将它分发给系统当前激活的应用程序窗口处理。

从前面14.4.1小节的Step 6可以知道，系统当前需要分发的键盘事件全部保存在InputDispatcher类的成员变量mInboundQueue所描述的一个待分发键盘事件队列中。第13行的if语句检查这个队列是否不为空。如果不为空，那么接下来第31行到第35行代码就会将保存在这个队列头部的键盘事件取出来，并且保存在InputDispatcher类的成员变量mPendingEvent中，以便接下来可以将它分发给系统当前激活的应用程序窗口处理。

第14行到第28行代码用来处理待分发键盘事件队列为空的情况。本来在这种情况下，InputDispatcher类的成员函数dispatchOnceInnerLocked是可以直接返回的，因为这时候没有待分发的键盘事件需要处理。但是由于InputDispatcher可能需要由自己来产生重复的键盘事件，如前面14.2.3小节的Step 2所述，因此，这时候InputDispatcher类的成员函数dispatchOnceInnerLocked还不能直接返回。

InputDispatcher类的成员变量mKeyRepeatState用来描述系统当前的重复键盘按键情况。如果它的成员变量lastKeyEntry不等于null，即第17行的if语句为true，那么接下来第18行到第24行代码就需要考虑产生一个重复的键盘事件分发给系统当前激活的应用程序窗口处理。第18行的if语句检查系统的当前时间是否已经到达下一个重复键盘事件的产生时间。如果已经到达了，那么第19行代码就会调用InputDispatcher类的成员函数synthesizeKeyRepeatLocked来合成一个重复的键盘事件，并且保存在InputDispatcher类的成员变量mPendingEvent中，以便接下来可以将它分发给系统当前激活的应用程序窗口处理；否则，第21行的if语句就会继续检查下一个重复键盘事件的产生时间是否小于输出参数nextWakeupTime的值。如果小于，那么第22行代码就会将输出参数nextWakeupTime的值修改为下一个重复键盘事件的产生时间，以便InputDispatpcher可以在指定的时间间隔内产生一个重复的键盘事件分发给系统当前激活的应用程序窗口处理。

假设第12行到第39行代码执行完成之后，InputDispatcher类的成员变量mPendingEvent不等于null，即它指向了一个即将要分发给系统当前激活的应用程序窗口处理的键盘事件；接下来我们就继续分析这个键盘事件的分发过程。

frameworks/base/libs/ui/InputDispatcher.cpp

```
40      DropReason dropReason = DROP_REASON_NOT_DROPPED;
41      ......
42
43      switch (mPendingEvent->type) {
44      ......
45      case EventEntry::TYPE_KEY: {
46          KeyEntry* typedEntry = static_cast<KeyEntry*>(mPendingEvent);
47          if (isAppSwitchDue) {
48              if (isAppSwitchKeyEventLocked(typedEntry)) {
49                  resetPendingAppSwitchLocked(true);
50                  ......
51              }
52              ......
53          }
54          done = dispatchKeyLocked(currentTime, typedEntry, keyRepeatTimeout,
55                  &dropReason, nextWakeupTime);
56          break;
```

```
57      }
58      ......
59      }
60      ......
61
62      if (done) {
63          ......
64
65          releasePendingEventLocked();
66          *nextWakeupTime = LONG_LONG_MIN;  // force next poll to wake up immediately
67      }
68  }
```

从前面的第6行代码可以知道，变量isAppSwitchDue用来描述系统的当前时间是否大于等于前面所发生的一个与应用程序窗口切换操作相关的键盘事件的分发时间。如果它的值等于true，即第47行的if语句为true，那么第48行的if语句就会继续判断当前所要分发的键盘事件是否就是一个与应用程序窗口切换操作相关的键盘事件。如果是，那么就说明即将分发的键盘事件就是前面所发生的一个与应用程序窗口切换操作相关的键盘事件。因此，第49行代码就会调用InputDispatcher类的成员函数resetPendingAppSwitchLocked将成员变量mAppSwitchDueTime的值设置为LONG_LONG_MAX，以便可以表示前面所发生的一个与应用程序窗口切换操作相关的键盘事件已经得到处理了。

接下来第54行和第55行代码调用InputDispatcher类的成员函数dispatchKeyLocked将KeyEntry结构体typedEntry所描述的一个键盘事件分发给系统当前激活的应用程序窗口处理。如果能够成功地分发出去，那么它的返回值就会等于true。在这种情况下，第65行代码首先会调用InputDispatcher类的成员函数releasePendingEventLocked将成员变量mPendingEvent的值设置为null，以便可以表示当前所需要分发的键盘事件已经成功地分发出去了；接着第66行代码再将输出参数nextWakeupTime 的值设置为LONG_LONG_MIN，以便InputDispatcher可以马上处理下一个需要分发的键盘事件。

接下来，我们继续分析InputDispatcher类的成员函数dispatchKeyLocked是如何将一个键盘事件分发给系统当前激活的应用程序窗口处理的。

Step 2：InputDispatcher.dispatchKeyLocked

frameworks/base/libs/ui/InputDispatcher.cpp

```
01 bool InputDispatcher::dispatchKeyLocked(
02          nsecs_t currentTime, KeyEntry* entry, nsecs_t keyRepeatTimeout,
03          DropReason* dropReason, nsecs_t* nextWakeupTime) {
04      ......
05
06      // Identify targets.
07      if (! mCurrentInputTargetsValid) {
08          int32_t injectionResult = findFocusedWindowTargetsLocked(currentTime,
09              entry, nextWakeupTime);
10
11          ......
12      }
13
14      // Dispatch the key.
15      dispatchEventToCurrentInputTargetsLocked(currentTime, entry, false);
16      return true;
17  }
```

InputDispatcher类的成员变量mCurrentInputTargetsValid用来描述InputDispatcher是否已经将系统当前激活的应用程序窗口保存在另外一个成员变量mCurrentInputTargets中了。如果还没有保存，那么接下来第8行和第9行代码就会首先调用InputDispatcher类的成员函数findFocusedWindowTargetsLocked将系统当前激活的应用程序窗口保存在成员变量mCurrentInputTargets中，接着第15行代码再调用InputDispatcher类的成员函数dispatchEventToCurrentInputTargetsLocked将参数entry所描述的一个键盘事

件分发给系统当前激活的应用程序窗口处理。

下面我们首先分析InputDispatcher类的成员函数findFocusedWindowTargetsLocked的实现，然后分析InputDispatcher类的成员函数dispatchEventToCurrentInputTargetsLocked的实现。

Step 3：InputDispatcher.findFocusedWindowTargetsLocked

frameworks/base/libs/ui/InputDispatcher.cpp

```
01 int32_t InputDispatcher::findFocusedWindowTargetsLocked(nsecs_t currentTime,
02         const EventEntry* entry, nsecs_t* nextWakeupTime) {
03     mCurrentInputTargets.clear();
04
05     int32_t injectionResult;
06
07     // If there is no currently focused window and no focused application
08     // then drop the event.
09     if (! mFocusedWindow) {
10         if (mFocusedApplication) {
11             ......
12             injectionResult = handleTargetsNotReadyLocked(currentTime, entry,
13                 mFocusedApplication, NULL, nextWakeupTime);
14             goto Unresponsive;
15         }
16
17         ......
18         injectionResult = INPUT_EVENT_INJECTION_FAILED;
19         goto Failed;
20     }
21
22     // Check permissions.
23     if (! checkInjectionPermission(mFocusedWindow, entry->injectionState)) {
24         injectionResult = INPUT_EVENT_INJECTION_PERMISSION_DENIED;
25         goto Failed;
26     }
27
28     // If the currently focused window is paused then keep waiting.
29     if (mFocusedWindow->paused) {
30         ......
31         injectionResult = handleTargetsNotReadyLocked(currentTime, entry,
32             mFocusedApplication, mFocusedWindow, nextWakeupTime);
33         goto Unresponsive;
34     }
35
36     // If the currently focused window is still working on previous events then keep waiting.
37     if (! isWindowFinishedWithPreviousInputLocked(mFocusedWindow)) {
38         ......
39         injectionResult = handleTargetsNotReadyLocked(currentTime, entry,
40             mFocusedApplication, mFocusedWindow, nextWakeupTime);
41         goto Unresponsive;
42     }
43
44     // Success!  Output targets.
45     injectionResult = INPUT_EVENT_INJECTION_SUCCEEDED;
46     addWindowTargetLocked(mFocusedWindow, InputTarget::FLAG_FOREGROUND, BitSet32(0));
47     ......
48
49     return injectionResult;
50 }
```

从前面14.3.3小节的Step 7可以知道，系统当前激活的应用程序窗口保存在InputDispatcher类的成员变量mFocusedWindow中。InputDispatcher类还有另外一个成员变量mFocusedApplication，它用来描述系统当前激活的应用程序。InputDispatcher类的成员函数findFocusedWindowTargetsLocked的目标就

是将系统当前激活的应用程序窗口mFocusedWindow封装成一个InputTarget对象，并且保存在成员变量mCurrentInputTargets中。

第9行的if语句首先检查输入管理器InputManager是否已经将系统当前激活的应用程序窗口注册到InputDispatcher中了。如果还没有注册，那么接下来第10行的if语句再检查输入管理器InputManager是否已经将系统当前激活的应用程序注册到InputDispatcher中了。如果也没有注册，那么参数entry所描述的一个键盘事件就分发失败了。如果输入管理器InputManager已经将系统当前激活的应用程序注册到InputDispatcher中，但是还没有将系统当前激活的应用程序窗口注册到InputDispatcher中，那么就说明系统当前激活的应用程序还没有启动完成，或者正处于无响应状态，这时候参数entry所描述的一个键盘事件也是无法分发出去的。

第23行的if语句调用InputDispatcher类的成员函数checkInjectionPermission来检查参数entry所描述的一个键盘事件是否是由应用程序注入进来的，即它不是由硬件产生的。在这种情况下，InputDispatcher类的成员函数checkInjectionPermission就会检查这个应用程序是否有权限向系统当前激活的应用程序窗口注入一个键盘事件。如果没有，那么参数entry所描述的一个键盘事件就分发失败了。

第29行的if语句检查系统当前激活的应用程序窗口是否处于Paused状态。如果是，那么就说明系统当前激活的应用程序窗口处于无响应状态，因此，参数entry所描述的一个键盘事件就会分发失败。

第37行的if语句调用InputDispatcher类的成员函数isWindowFinishedWithPreviousInputLocked来检查系统当前激活的应用程序窗口是否已经处理完成InputDispatcher上一次分发给它的一个键盘事件了。如果还没有处理完成，那么InputDispatcher就不能继续将下一个键盘事件分发给它处理，即不能将参数entry所描述的一个键盘事件分发给它处理。

通过了前面四个if语句的合法性检查之后，接下来第46行代码就会调用InputDispatcher类的成员函数addWindowTargetLocked将系统当前激活的应用程序窗口mFocusedWindow封装成一个InputTarget对象，并且保存在InputDispatcher类的成员变量mCurrentInputTargets中。

InputDispatcher类的成员函数addWindowTargetLocked的实现如下所示。

frameworks/base/libs/ui/InputDispatcher.cpp

```
01 void InputDispatcher::addWindowTargetLocked(const InputWindow* window, int32_t targetFlags,
02         BitSet32 pointerIds) {
03     mCurrentInputTargets.push();
04
05     InputTarget& target = mCurrentInputTargets.editTop();
06     target.inputChannel = window->inputChannel;
07     target.flags = targetFlags;
08     target.xOffset = - window->frameLeft;
09     target.yOffset = - window->frameTop;
10     target.pointerIds = pointerIds;
11 }
```

InputDispatcher类的成员变量mCurrentInputTargets是一个类型为InputTarget的Vector，第3行和第5行代码首先在它的末尾创建一个InputTarget对象，接着第6行到第10行代码再使用参数window所描述的一个InputWindow对象的信息来初始化这个InputTarget对象的各个成员变量，其中，最重要的就是初始化它的成员变量inputChannel。

从前面的调用过程可以知道，参数window是用来描述系统当前激活的应用程序窗口的，而它的成员变量inputChannel是用来描述一个Server端InputChannel的。接下来我们就可以看到，InputDispatcher就是通过这个Server端InputChannel将键盘事件分给系统当前激活的应用程序程序窗口处理的。

这一步执行完成之后，返回到前面的Step 2中，即InputDispatcher类的成员函数dispatchKeyLocked中，接下来它就会调用InputDispatcher类的成员函数dispatchEventToCurrentInputTargetsLocked向系统当前激活的应用程序窗口分发一个键盘事件。

Step 4：InputDispatcher.dispatchEventToCurrentInputTargetsLocked

frameworks/base/libs/ui/InputDispatcher.cpp

```
01 void InputDispatcher::dispatchEventToCurrentInputTargetsLocked(nsecs_t currentTime,
02         EventEntry* eventEntry, bool resumeWithAppendedMotionSample) {
03     ......
04
05     for (size_t i = 0; i < mCurrentInputTargets.size(); i++) {
06         const InputTarget& inputTarget = mCurrentInputTargets.itemAt(i);
07
08         ssize_t connectionIndex = getConnectionIndexLocked(inputTarget.inputChannel);
09         if (connectionIndex >= 0) {
10             sp<Connection> connection = mConnectionsByReceiveFd.valueAt(connectionIndex);
11             prepareDispatchCycleLocked(currentTime, connection, eventEntry, & inputTarget,
12                 resumeWithAppendedMotionSample);
13         }
14         ......
15     }
16 }
```

系统当前需要接收键盘事件的应用程序窗口都保存在InputDispatcher类的成员变量mCurrentInputTargets中，第5行到第15行的for循环依次向它们分发一个键盘事件。从前面Step 3的内容可以知道，InputDispatcher类的成员变量mCurrentInputTargets包含了系统当前激活的应用程序窗口，接下来我们就主要分析InputDispatcher向它分发键盘事件的过程。

InputDispatcher首先要找到与系统当前激活的应用程序窗口所关联的一个Connection对象，然后才可以通过这个Connection对象向系统当前激活的应用程序窗口分发一个键盘事件。从前面14.3.2小节的内容可以知道，这个Connection对象是与一个Server端InputChannel关联在一起的，并且保存在InputDispatcher类的成员变量mConnectionsByReceiveFd中，用来在InputDispatcher和系统当前激活的应用程序窗口之间建立一个前向连接。

由于前面我们已经知道了与系统当前激活的应用程序窗口所关联的一个Server端InputChannel，因此，第8行和第10行代码就可以根据它在InputDispatcher类的成员变量mConnectionsByReceiveFd中找到与它所关联的一个Connection对象。有了这个Connection对象之后，接下来第11行和第12行代码就调用InputDispatcher类的成员函数prepareDispatchCycleLocked向系统当前激活的应用程序窗口分发一个键盘事件。

Step 5：InputDispatcher.prepareDispatchCycleLocked

frameworks/base/libs/ui/InputDispatcher.cpp

```
01 void InputDispatcher::prepareDispatchCycleLocked(nsecs_t currentTime,
02         const sp<Connection>& connection, EventEntry* eventEntry, const InputTarget* inputTarget,
03         bool resumeWithAppendedMotionSample) {
04     ......
05
06     bool wasEmpty = connection->outboundQueue.isEmpty();
07     ......
08
09     // This is a new event.
10     // Enqueue a new dispatch entry onto the outbound queue for this connection.
11     DispatchEntry* dispatchEntry = mAllocator.obtainDispatchEntry(eventEntry, // increments ref
12         inputTarget->flags, inputTarget->xOffset, inputTarget->yOffset);
13     ......
14
15     // Enqueue the dispatch entry.
16     connection->outboundQueue.enqueueAtTail(dispatchEntry);
```

```
17
18      // If the outbound queue was previously empty, start the dispatch cycle going.
19      if (wasEmpty) {
20          ......
21
22          startDispatchCycleLocked(currentTime, connection);
23      }
24  }
```

每一个Connection对象内部都有一个独立的待分发键盘事件队列outboundQueue，保存在它里面的键盘事件需要依次分发给与该Connection对象所关联的一个应用程序窗口处理，即只有上一个键盘事件被目标应用程序窗口处理完成之后，下一个键盘事件才能继续分发给目标应用程序窗口处理。

第6行代码首先检查与系统当前激活的应用程序窗口所关联的一个待分发键盘事件队列是否为空。如果这个队列不为空，那么就说明上一次分发给系统当前激活的应用程序窗口的键盘事件还没有被处理完成。在这种情况下，当前正在处理的键盘事件，即参数eventEntry所描述的一个键盘事件，就不能马上分发给系统当前激活的应用程序窗口处理。

第11行和第12行代码首先将参数eventEntry所描述的一个键盘事件封装成一个DispatchEntry结构体；然后第16行代码将这个DispatchEntry结构体添加到与系统当前激活的应用程序窗口所关联的一个待分发键盘事件队列的末尾；最后第19行的if语句检查这个队列之前是否是空的。如果这个队列之前是空的，那么就说明系统当前激活的应用程序窗口已经处理完成前面所有分发给它的键盘事件了。因此，接下来第22行代码就会调用InputDispatcher类的成员函数startDispatchCycleLocked向它分发当前正在处理的键盘事件，即参数eventEntry所描述的一个键盘事件。

Step 6：InputDispatcher.startDispatchCycleLocked

frameworks/base/libs/ui/InputDispatcher.cpp

```
01 void InputDispatcher::startDispatchCycleLocked(nsecs_t currentTime,
02     const sp<Connection>& connection) {
03     ......
04
05     DispatchEntry* dispatchEntry = connection->outboundQueue.headSentinel.next;
06
07     // Mark the dispatch entry as in progress.
08     dispatchEntry->inProgress = true;
09
10     // Update the connection's input state.
11     EventEntry* eventEntry = dispatchEntry->eventEntry;
12     ......
13
14     // Publish the event.
15     status_t status;
16     switch (eventEntry->type) {
17     case EventEntry::TYPE_KEY: {
18         KeyEntry* keyEntry = static_cast<KeyEntry*>(eventEntry);
19
20         // Apply target flags.
21         int32_t action = keyEntry->action;
22         int32_t flags = keyEntry->flags;
23
24         // Publish the key event.
25         status = connection->inputPublisher.publishKeyEvent(keyEntry->deviceId, keyEntry->source,
26             action, flags, keyEntry->keyCode, keyEntry->scanCode,
27             keyEntry->metaState, keyEntry->repeatCount, keyEntry->downTime,
28             keyEntry->eventTime);
29
30         ......
```

```
31          break;
32      }
33      ......
34      }
35
36      // Send the dispatch signal.
37      status = connection->inputPublisher.sendDispatchSignal();
38
39      ......
40  }
```

当前正在处理的键盘事件保存在参数connection内部的一个待分发键盘事件队列outboundQueue的头部，第5行代码首先将它取出来，并且保存在DispatchEntry结构体dispatchEntry中；接着第8行代码将这个DispatchEntry结构体的成员变量inProgress的值设置为true，表示它所描述的一个键盘事件正处于分发过程中。

DispatchEntry结构体dispatchEntry所描述的键盘事件的数据保存在它的成员变量eventEntry中，第11行代码将它取出来，并且保存在EventEntry结构体eventEntry中。由于EventEntry结构体eventEntry所描述的是一个键盘事件，即它的成员变量type的值等于EventEntry::TYPE_KEY，因此，接下来第18行代码就可以安全地将它转换成一个KeyEntry结构体keyEntry。通过KeyEntry结构体keyEntry，我们就可以获得所要分发的键盘事件的所有数据了。

在前面14.3.2小节的Step 4中提到，一个Connection对象实际上是通过它内部的一个InputPublisher对象来向系统当前激活的应用程序窗口分发键盘事件的。因此，第25行到第28行代码首先将即将要分发的键盘事件的数据写入到参数connection内部的一个InputPublisher对象中，接着第37行代码再调用这个InputPublisher对象的成员函数sendDispatchSignal来通知系统当前激活的应用程序窗口来接收一个键盘事件。

接下来，我们首先分析InputDispatcher将一个键盘事件的数据写入到一个InputPublisher对象中的过程，即InputPublisher类的成员函数publishKeyEvent的实现；然后分析InputDispatcher通知系统当前激活的应用程序窗口接收一个键盘事件的过程，即InputPublisher类的成员函数sendDispatchSignal的实现。

Step 7：InputPublisher.publishKeyEvent

frameworks/base/libs/ui/InputTransport.cpp

```
01 status_t InputPublisher::publishKeyEvent(
02      int32_t deviceId,
03      int32_t source,
04      int32_t action,
05      int32_t flags,
06      int32_t keyCode,
07      int32_t scanCode,
08      int32_t metaState,
09      int32_t repeatCount,
10      nsecs_t downTime,
11      nsecs_t eventTime) {
12      ......
13
14      status_t result = publishInputEvent(AINPUT_EVENT_TYPE_KEY, deviceId, source);
15      if (result < 0) {
16          return result;
17      }
18
19      mSharedMessage->key.action = action;
20      mSharedMessage->key.flags = flags;
21      mSharedMessage->key.keyCode = keyCode;
22      mSharedMessage->key.scanCode = scanCode;
```

```
23      mSharedMessage->key.metaState = metaState;
24      mSharedMessage->key.repeatCount = repeatCount;
25      mSharedMessage->key.downTime = downTime;
26      mSharedMessage->key.eventTime = eventTime;
27      return OK;
28  }
```

在前面14.3.2小节的Step 4中提到，每一个InputPublisher对象内部都引用了一块匿名共享内存，这块匿名共享内存是在创建与它所关联的一个Server端InputChannel时创建的，如前面14.3.1小节的Step 6所示。InputPublisher类的成员函数publishKeyEvent的目标就是将一个即将要分发的键盘事件的数据写入到一个InputPublisher对象内部所引用的一块匿名共享内存中，以便可以传递给系统当前激活的应用程序窗口访问。

在前面14.3.2小节的Step 4中还提到，每一个InputPublisher对象内部所引用的一个匿名共享内存块都是使用一个InputMessage结构体来描述的，这个InputMessage结构体就保存在这些InputPublisher对象的成员变量mSharedMessage中。因此，第19行到26行代码就可以通过InputPublisher类的成员变量mSharedMessage将一个即将要分发的键盘事件的数据写入到一个InputPublisher对象内部所引用的一块匿名共享内存中。

从前面第6章的内容可以知道，在向一块匿名共享内存写入数据之前，需要对它进行锁定，避免它在使用的过程中被内存管理系统回收。锁定一个InputPublisher对象内部所引用的一个匿名共享内存块的操作是由InputPublisher类的成员函数publishInputEvent来执行的，它的实现如下所示。

frameworks/base/libs/ui/InputTransport.cpp

```
01 status_t InputPublisher::publishInputEvent(
02         int32_t type,
03         int32_t deviceId,
04         int32_t source) {
05     if (mPinned) {
06         ......
07         return INVALID_OPERATION;
08     }
09     ......
10
11     int ashmemFd = mChannel->getAshmemFd();
12     int result = ashmem_pin_region(ashmemFd, 0, 0);
13     if (result < 0) {
14         ......
15         return UNKNOWN_ERROR;
16     }
17
18     mPinned = true;
19     ......
20
21     mSharedMessage->consumed = false;
22     ......
23     return OK;
24 }
```

InputPublisher类的成员变量mPinned是一个布尔值，用来描述一个InputPublisher对象内部所引用的一个匿名共享内存块是否正在使用中。如果它的值等于true，那么就说明这个匿名共享内存块正处于使用的过程中，因此，我们就不能向它写入一个即将要分发的键盘事件的数据。

InputPublisher类的成员变量mChannel指向了一个Server端InputChannel，一个InputPublisher对象内部所引用的一个匿名共享内存块的文件描述符就保存这个Server端InputChannel中。第11行代码首先调用InputPublisher类的成员变量mChannel的成员函数getAshmemFd来获得一个InputPublisher对象内部所引用的一个匿名共享内存块的文件描述符，接着第12行代码再调用运行时库cutils所提供的匿名共享内存访问接口ashmem_pin_region来锁定这个匿名共享内存块。锁定成功之后，接下来第18行代码首先

将这个InputPublisher对象的成员变量mPinned的值设置为true，以便可以表示它内部所引用的一个匿名共享内存块正在使用中；接着第21行代码再将这个InputPublisher对象的成员变量mSharedMessage所描述的一个InputMessage结构体的成员变量consumed的值设置为false，以便可以表示保存在它里面的键盘事件数据还没有被目标应用程序窗口读取。

这一步执行完成之后，返回到前面的Step 6中，即InputDispatcher类的成员函数startDispatchCycleLocked中，接下来就会调用成员函数sendDispatchSignal来通知系统当前激活的应用程序窗口接收一个键盘事件。

Step 8：InputPublisher.sendDispatchSignal

frameworks/base/libs/ui/InputTransport.cpp

```
1 status_t InputPublisher::sendDispatchSignal() {
2     ......
3
4     return mChannel->sendSignal(INPUT_SIGNAL_DISPATCH);
5 }
```

第4行调用InputPublisher类的成员变量mChannel所指向的一个Server端InputChannel的成员函数sendSignal来通知系统当前激活的应用程序窗口接收一个键盘事件。

Step 9：InputChannel.sendSignal

frameworks/base/libs/ui/InputTransport.cpp

```
01 status_t InputChannel::sendSignal(char signal) {
02     ssize_t nWrite;
03     do {
04         nWrite = ::write(mSendPipeFd, & signal, 1);
05     } while (nWrite == -1 && errno == EINTR);
06
07     if (nWrite == 1) {
08         ......
09         return OK;
10     }
11
12     return -errno;
13 }
```

从前面14.3.1小节的Step 6可以知道，Server端InputChannel的成员变量mSendPipeFd用来描述一个前向管道的写端文件描述符，第3行到第5行代码通过这个文件描述符向这个前向管道写入一个字符signal（INPUT_SIGNAL_DISPATCH），以便可以通知系统当前激活的应用程序窗口来接收一个键盘事件。

从前面14.3.4小节的Step 3可以知道，当InputDispatcher通过一个Server端InputChannel向系统当前激活的应用程序窗口分发一个键盘事件时，系统当前激活的应用程序窗口所运行在的应用程序进程的主线程首先会获得这个键盘事件，然后它再将这个键盘事件封装成一个键盘消息交给系统当前激活的应用程序窗口来处理。

接下来，我们就继续分析系统当前激活的应用程序窗口获得一个键盘消息的过程。

14.4.3　系统当前激活的应用程序窗口获得键盘消息

从前面14.3.4小节的Step 3可以知道，当InputDispatcher向系统当前激活的应用程序窗口分发一个键盘事件时，系统当前激活的应用程序窗口所运行在的应用程序进程的主线程就会调用NativeInputQueue类的静态成员函数handleReceiveCallback来接收这个键盘事件。

一个应用程序主线程接收到一个键盘事件之后，首先会将这个键盘事件封装成一个键盘消息，然后再发送到它的消息队列中，以便可以分发给系统当前激活的应用程序窗口来处理。这个过程一共可以分为18个步骤，下面我们就详细分析每一个步骤。

第1步到第8步如图14-15所示。

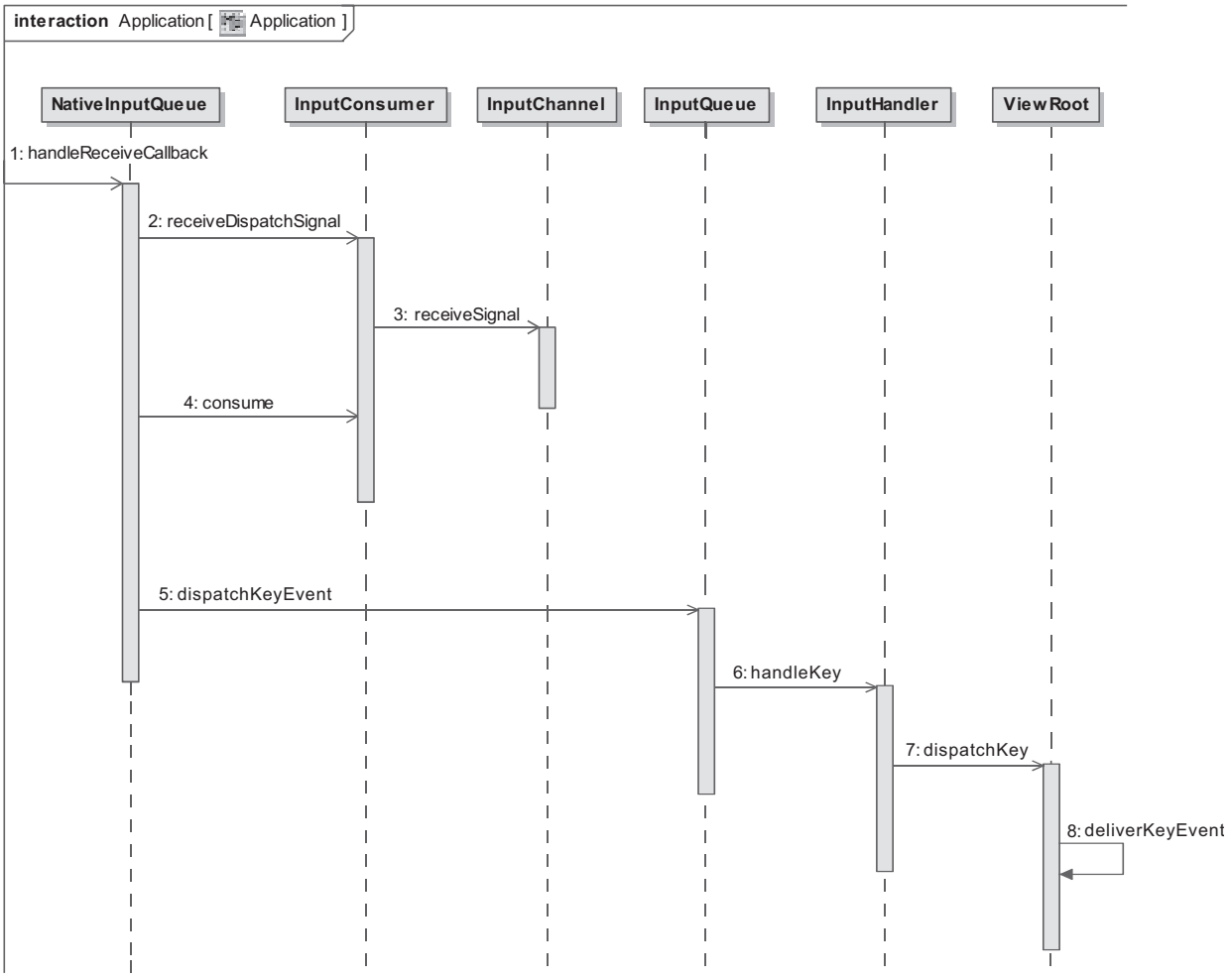


图14-15　系统当前激活的应用程序窗口获得键盘消息的过程——Step 1至Step 8

Step 1：NativeInputQueue.handleReceiveCallback

frameworks/base/core/jni/android_view_InputQueue.cpp

```
01 int NativeInputQueue::handleReceiveCallback(int receiveFd, int events, void* data) {
02     NativeInputQueue* q = static_cast<NativeInputQueue*>(data);
03     JNIEnv* env = AndroidRuntime::getJNIEnv();
04
05     sp<Connection> connection;
06     InputEvent* inputEvent;
07     jobject inputHandlerObjLocal;
08     jlong finishedToken;
09     { // acquire lock
10         AutoMutex _l(q->mLock);
11
12         ssize_t connectionIndex = q->mConnectionsByReceiveFd.indexOfKey(receiveFd);
13         ......
14
15         connection = q->mConnectionsByReceiveFd.valueAt(connectionIndex);
```

```
16          ......
17
18          status_t status = connection->inputConsumer.receiveDispatchSignal();
19          if (status) {
20              ......
21              return 0; // remove the callback
22          }
23          ......
24
25          status = connection->inputConsumer.consume(& connection->inputEventFactory, & inputEvent);
26          ......
27
28          finishedToken = generateFinishedToken(receiveFd, connection->id, connection->messageSeqNum);
29
30          inputHandlerObjLocal = env->NewLocalRef(connection->inputHandlerObjGlobal);
31      } // release lock
32      ......
33
34      int32_t inputEventType = inputEvent->getType();
35
36      jobject inputEventObj;
37      jmethodID dispatchMethodId;
38      switch (inputEventType) {
39      case AINPUT_EVENT_TYPE_KEY:
40          ......
41          inputEventObj = android_view_KeyEvent_fromNative(env,
42                                  static_cast<KeyEvent*>(inputEvent));
43          dispatchMethodId = gInputQueueClassInfo.dispatchKeyEvent;
44          break;
45      }
46      ......
47      }
48      ......
49
50      env->CallStaticVoidMethod(gInputQueueClassInfo.clazz,
51                              dispatchMethodId, inputHandlerObjLocal, inputEventObj,
52                              jlong(finishedToken));
53      ......
54
55      return 1;
56  }
```

从前面14.3.4小节的Step 3可以知道，参数data指向了一个NativeInputQueue对象，因此，第2行代码就可以安全地将它转换为一个NativeInputQueue对象，并且保存在变量q中。

从前面14.3.4小节的Step 3还可以知道，参数receiveFd是用来描述一个Client端InputChannel内部的一个前向管道的读端文件描述符的，第12行和第15行代码通过它在NativeInputQueue对象q的成员变量mConnectionsByReceiveFd中找到一个对应的Connection对象connection。这个Connection对象是用来在InputDisptacher和系统当前激活的应用程序窗口之间建立连接的。有了这个Connection对象之后，系统当前激活的应用程序窗口所运行在的应用程序进程的主线程就可以接收到InputDisptacher分发过来的键盘事件了。

每一个Connection对象内部都有一个InputConsumer对象，通过它就可以将InputDisptacher分发给系统当前激活的应用程序窗口的键盘事件数据读取出来，这是通过调用它的成员函数consume来实现的，如第25行代码所示。不过，在调用一个InputConsumer对象的成员函数consume来读取InputDisptacher分发过来的一个键盘事件数据之前，首先要调用它的成员函数receiveDispatchSignal来确认InputDisptacher的确是向系统当前激活的应用程序窗口分发了一个键盘事件，如第18行代码所

示。读取出来的键盘事件数据保存在一个InputEvent对象inputEvent中。

第28行代码调用NativeInputQueue类的静态成员函数generateFinishedToken为前面所接收到的一个键盘事件生成一个令牌finishedToken。这个令牌包含了三个信息，第一个信息是参数receiveFd所描述的一个前向管道的读端文件描述符；第二个和第三个信息是前面所获得的一个Connection对象connection的成员变量id 和messageSeqNum的值。系统当前激活的应用程序窗口在处理完成一个键盘事件之后，就会根据这个令牌找回前面所获得的一个Connection对象connection，以便可以通过这个Connection对象向InputDispatcher发送一个键盘事件处理完成通知。

NativeInputQueue类的静态成员函数generateFinishedToken的实现如下所示。

frameworks/base/core/jni/android_view_InputQueue.cpp

```
1 jlong NativeInputQueue::generateFinishedToken(int32_t receiveFd, uint16_t connectionId,
2         uint16_t messageSeqNum) {
3     return (jlong(receiveFd) << 32) | (jlong(connectionId) << 16) | jlong(messageSeqNum);
4 }
```

第3行代码分别将参数receiveFd、connectionId和messageSeqNum的值保存在一个长整数的高32位、中间16位和低16位中，并且将这个长整数返回给调用者。

返回到NativeInputQueue类的静态成员函数handleReceiveCallback中，我们继续分析它的实现。

从前面14.3.1小节的Step 1可以知道，前面所获得的一个Connection对象connection的成员变量inputHandlerObjGlobal指向了一个Java层的InputHandler对象。第30行代码获得它的一个引用，并且保存在变量inputHandlerObjLocal中，接下来就会将它返回到Java层中，并且调用它的成员函数handleKey来进一步处理这里所接收到的一个键盘事件。

由于InputEvent对象inputEvent所描述的是一个键盘事件，因此，第41行和第42行代码就可以安全地将它转换为一个C++层的KeyEvent对象，接着再将这个C++层的KeyEvent对象封装成一个Java层的KeyEvent对象inputEventObj，以便可以将InputEvent对象inputEvent所描述的一个键盘事件返回给运行在Java层中的应用程序窗口处理。

gInputQueueClassInfo是一个全局变量，它的成员变量clazz指向了Java层的InputQueue类；而成员变量dispatchKeyEvent指向了Java层的InputQueue类的静态成员函数dispatchKeyEvent，通过它就可以调用Java层的InputQueue类的静态成员函数dispatchKeyEvent来进一步处理前面所获得的一个键盘事件inputEventObj，如第50行到第52行代码所示。

接下来，我们首先分析InputConsumer类的成员函数receiveDispatchSignal和consume的实现，以便可以了解它们是如何确认和获取InputDispatcher分发过来的一个键盘事件的；然后分析Java层的InputQueue类的静态成员函数dispatchKeyEvent的实现，以便可以了解它是如何处理InputDispatcher分发过来的一个键盘事件的。

Step 2：InputConsumer.receiveDispatchSignal

frameworks/base/libs/ui/InputTransport.cpp

```
01 status_t InputConsumer::receiveDispatchSignal() {
02     ......
03
04     char signal;
05     status_t result = mChannel->receiveSignal(& signal);
06     if (result) {
07         return result;
08     }
09     if (signal != INPUT_SIGNAL_DISPATCH) {
10         ......
11         return UNKNOWN_ERROR;
12     }
13     return OK;
14 }
```

InputConsumer类的成员变量mChannel指向了一个Client端InputChannel，第5行代码首先调用它的成员函数receiveSignal从它内部的一个前向管道中读出一个字符，接着第9行的if语句检查这个字符的值是否等于INPUT_SIGNAL_DISPATCH。从前面14.4.2小节的Step 9可以知道，InputDispatcher向这个前向管道写入的字符的值正好等于INPUT_SIGNAL_DISPATCH，因此，接下来第13行代码就会返回一个OK值给调用者，表示InputDispatcher确实是向系统当前激活的应用程序窗口分发了一个键盘事件。

下面我们继续分析Client端InputChannel类的成员函数receiveSignal是如何从一个前向管道中读出一个字符的。

Step 3：InputChannel. receiveSignal

frameworks/base/libs/ui/InputTransport.cpp

```
01 status_t InputChannel::receiveSignal(char* outSignal) {
02     ssize_t nRead;
03     do {
04         nRead = ::read(mReceivePipeFd, outSignal, 1);
05     } while (nRead == -1 && errno == EINTR);
06
07     if (nRead == 1) {
08         ......
09         return OK;
10     }
11
12     ......
13     return -errno;
14 }
```

从前面14.3.1小节的Step 6可以知道，Client端InputChannel的成员变量mReceivePipeFd是用来描述一个前向管道的读端文件描述符的。因此，第3行到第5行代码就可以通过这个文件描述符从它所描述的一个前向管道中读出一个字符值，并且保存在输出参数outSignal中。

这一步执行完成之后，返回到前面的Step 1中，即NativeInputQueue类的静态成员函数handleReceiveCallback中，接下来就会继续调用InputConsumer类的成员函数consume来读取InputDispatcher分发过来的一个键盘事件的数据。

Step 4：InputConsumer.consume

frameworks/base/libs/ui/InputTransport.cpp

```
01 status_t InputConsumer::consume(InputEventFactoryInterface* factory, InputEvent** outEvent) {
02     ......
03
04     *outEvent = NULL;
05
06     int ashmemFd = mChannel->getAshmemFd();
07     int result = ashmem_pin_region(ashmemFd, 0, 0);
08     if (result != ASHMEM_NOT_PURGED) {
09         ......
10         return UNKNOWN_ERROR;
11     }
12     ......
13
14     if (mSharedMessage->consumed) {
15         ......
16         return INVALID_OPERATION;
17     }
18     ......
19
20     mSharedMessage->consumed = true;
21
```

```
22      switch (mSharedMessage->type) {
23      case AINPUT_EVENT_TYPE_KEY: {
24          KeyEvent* keyEvent = factory->createKeyEvent();
25          if (! keyEvent) return NO_MEMORY;
26
27          populateKeyEvent(keyEvent);
28
29          *outEvent = keyEvent;
30          break;
31      }
32      ......
33      }
34
35      return OK;
36  }
```

InputConsumer类的成员函数consume与InputPublisher类的成员函数publishKeyEvent所执行的操作刚好是相反的，前者从一个匿名共享内存块中读出一个键盘事件数据，而后者往一个匿名共享内存块中写入一个键盘事件数据。

InputConsumer类的成员变量mChannel指向了一个Client端InputChannel，一个InputConsumer对象内部所引用的一个匿名共享内存块的文件描述符就保存在这个Client端InputChannel中。第6行代码首先调用InputConsumer类的成员变量mChannel的成员函数getAshmemFd来获得一个InputConsumer对象内部所引用的一个匿名共享内存块的文件描述符，接着第7行代码再调用运行时库cutils所提供的匿名共享内存访问接口ashmem_pin_region来检查这个匿名共享内存块是否已经被内存管理系统回收。如果已经被回收，即第8行的if语句为true，那么第10行代码就直接返回一个错误码UNKNOWN_ERROR给调用者。

在前面14.3.4小节的Step 3中提到，InputConsumer类内部所引用的一个匿名共享内存块是使用一个InputMessage结构体来描述的，第14行的if语句通过这个InputMessage结构体的成员变量consumed来检查保存在一个InputConsumer对象内部所引用的一个匿名共享内存块中的键盘事件数据是否已经被读取出来了。如果已经被读取出来了，那么就不能继续对保存在这个匿名共享内存块中的键盘事件数据进行读取了，因为它的内容可能已经发生了变化。因此，第16行代码就直接返回一个错误码INVALID_OPERATION给调用者。

第20行代码将当前正在处理的一个InputConsumer对象的成员变量mSharedMessage所描述的一个InputMessage结构体的成员变量consumed的值设置为true，因为接下来马上就要将保存在它里面的键盘事件数据读取出来了。

第24行代码首先创建了一个未初始化的KeyEvent对象，用来描述一个键盘事件；接着第27行代码再调用InputConsumer类的成员函数populateKeyEvent来对这个KeyEvent对象进行初始化，即使用保存在一个匿名共享内存块中的键盘事件数据来初始化这个KeyEvent对象，如下所示。

frameworks/base/libs/ui/InputTransport.cpp

```
01 void InputConsumer::populateKeyEvent(KeyEvent* keyEvent) const {
02      keyEvent->initialize(
03              mSharedMessage->deviceId,
04              mSharedMessage->source,
05              mSharedMessage->key.action,
06              mSharedMessage->key.flags,
07              mSharedMessage->key.keyCode,
08              mSharedMessage->key.scanCode,
09              mSharedMessage->key.metaState,
10              mSharedMessage->key.repeatCount,
11              mSharedMessage->key.downTime,
12              mSharedMessage->key.eventTime);
13  }
```

这一步执行完成之后，返回到前面的Step 1中，即NativeInputQueue类的静态成员函数handleReceiveCallback中，这时候系统当前激活的应用程序窗口所运行在的应用程序进程的主线程就获得InputDispatcher分发过来的一个键盘事件了，接下来它就会调用Java层的InputQueue类的静态成员函数dispatchKeyEvent来处理这个键盘事件。

Step 5：InputQueue.dispatchKeyEvent

frameworks/base/core/java/android/view/InputQueue.java

```
01 public final class InputQueue {
02     ......
03
04     private static void dispatchKeyEvent(InputHandler inputHandler,
05             KeyEvent event, long finishedToken) {
06         Runnable finishedCallback = FinishedCallback.obtain(finishedToken);
07         inputHandler.handleKey(event, finishedCallback);
08     }
09
10     ......
11 }
```

第6行代码使用参数finishedToken来创建一个类型为FinishedCallback的回调接口finishedCallback。系统当前激活的应用程序窗口处理完成参数event所描述的一个键盘事件之后，就会通过这个回调接口向InputDispatcher发送一个键盘事件处理完成通知。

参数inputHandler指向了一个InputHandler对象，第7行代码调用它的成员函数handleKey来进一步处理参数event所描述的一个键盘事件。

Step 6：InputHandler.handleKey

frameworks/base/core/java/android/view/ViewRoot.java

```
01 public final class ViewRoot extends Handler implements ViewParent,
02         View.AttachInfo.Callbacks {
03     ......
04
05     private final InputHandler mInputHandler = new InputHandler() {
06         public void handleKey(KeyEvent event, Runnable finishedCallback) {
07             startInputEvent(finishedCallback);
08             dispatchKey(event, true);
09         }
10         ......
11     };
12
13     ......
14 }
```

第7行代码首先调用外部类ViewRoot的成员函数startInputEvent将参数finishedCallback所描述的一个FinishedCallback对象保存在它的成员变量mFinishedCallback中，如下所示。

frameworks/base/core/java/android/view/ViewRoot.java

```
01 public final class ViewRoot extends Handler implements ViewParent,
02         View.AttachInfo.Callbacks {
03     ......
04
05     private void startInputEvent(Runnable finishedCallback) {
06         ......
07
08         mFinishedCallback = finishedCallback;
09     }
10
11     ......
12 }
```

回到InputHandler类的成员函数handleKey中，接着第8行代码继续调用外部类ViewRoot的成员函数dispatchKey来处理参数event所描述的一个键盘事件。

Step 7：ViewRoot.dispatchKey

frameworks/base/core/java/android/view/ViewRoot.java

```
01 public final class ViewRoot extends Handler implements ViewParent,
02         View.AttachInfo.Callbacks {
03     ......
04
05     private void dispatchKey(KeyEvent event, boolean sendDone) {
06         ......
07
08         Message msg = obtainMessage(DISPATCH_KEY);
09         msg.obj = event;
10         msg.arg1 = sendDone ? 1 : 0;
11
12         ......
13
14         sendMessageAtTime(msg, event.getEventTime());
15     }
16
17     ......
18 }
```

从前面的Step 6可以知道，参数sendDone的值等于true，表示系统当前激活的应用程序窗口在处理完成参数event所描述的一个键盘事件之后，需要向InputDispatcher发送一个键盘事件处理完成通知。

第8行到第10行代码首先将参数event所描述的一个键盘事件封装成一个键盘消息，即一个类型为DISPATCH_KEY的消息，接着第14行代码再将这个键盘消息发送到系统当前激活的应用程序窗口所运行在的应用程序进程的主线程的消息队列中。这个键盘消息最终是由ViewRoot类的成员函数deliverKeyEvent来处理的。

Step 8：ViewRoot.deliverKeyEvent

frameworks/base/core/java/android/view/ViewRoot.java

```
01 public final class ViewRoot extends Handler implements ViewParent,
02         View.AttachInfo.Callbacks {
03     ......
04
05     final InputMethodCallback mInputMethodCallback;
06     ......
07
08     View mView;
09     ......
10
11     boolean mLastWasImTarget;
12     ......
13
14     private void deliverKeyEvent(KeyEvent event, boolean sendDone) {
15         // If mView is null, we just consume the key event because it doesn't
16         // make sense to do anything else with it.
17         boolean handled = mView != null
18             ? mView.dispatchKeyEventPreIme(event) : true;
19         if (handled) {
20             if (sendDone) {
21                 finishInputEvent();
22             }
23             return;
```

```
24          }
25
26          // If it is possible for this window to interact with the input
27          // method window, then we want to first dispatch our key events
28          // to the input method.
29          if (mLastWasImTarget) {
30              InputMethodManager imm = InputMethodManager.peekInstance();
31              if (imm != null && mView != null) {
32                  ......
33
34                  imm.dispatchKeyEvent(mView.getContext(), seq, event,
35                      mInputMethodCallback);
36                  return;
37              }
38          }
39          deliverKeyEventToViewHierarchy(event, sendDone);
40      }
41
42      ......
43  }
```

在前面的14.1小节中提到，ViewRoot类的成员变量mView指向了一个DecorView对象，它是与系统当前激活的应用程序窗口关联在一起的，第17行和第18行代码首先调用它的成员函数dispatchKeyEventPreIme来处理参数event所描述的一个键盘事件。

DecorView类的成员函数dispatchKeyEventPreIme的返回值是一个布尔值。如果它的值等于true，那么ViewRoot类的成员函数就结束对参数event所描述的一个键盘事件的处理，如第20行到第23行代码所示。

注意

第20行的if语句用来检查参数sendDone的值是否等于true。如果等于true，那么接下来第21行代码就会调用ViewRoot类的成员函数finishInputEvent向InputDispatcher发送一个键盘事件处理完成通知。

假设DecorView类的成员函数dispatchKeyEventPreIme的返回值等于false，接下来第29行的if语句就会继续检查是否需要将参数event所描述的一个键盘事件分发给输入法管理器处理，这取决于系统当前激活的应用程序窗口是否支持输入法。

ViewRoot类的成员变量mLastWasImTarget用来描述系统当前激活的应用程序窗口是否支持输入法。如果支持，那么它的值就等于true，即第29行的if语句为true，接下来第30行代码就会首先获得系统当前所使用的输入法管理器，然后第34行和第35行代码再调用它的成员函数dispatchKeyEvent来处理参数event所描述的一个键盘事件。

ViewRoot类的成员变量mInputMethodCallback是一个输入法回调接口，输入法管理器处理完成参数event所描述的一个键盘事件之后，就会调用它的成员函数finishedEvent向ViewRoot类发送一个键盘事件处理完成通知。ViewRoot类接收到这个通知之后，接下来就会调用它的成员函数deliverKeyEventToViewHierarchy来继续处理参数event所描述的一个键盘事件。

如果系统当前激活的应用程序窗口不支持输入法，即第29行的if语句为false，那么接下来第39行代码就会直接调用ViewRoot类的成员函数deliverKeyEventToViewHierarchy来处理参数event所描述的一个键盘事件。

ViewRoot类的成员函数deliverKeyEventToViewHierarchy的作用是将参数event所描述的一个键盘事件分发给系统当前激活的应用程序窗口处理，它的调用过程如图14–16所示。

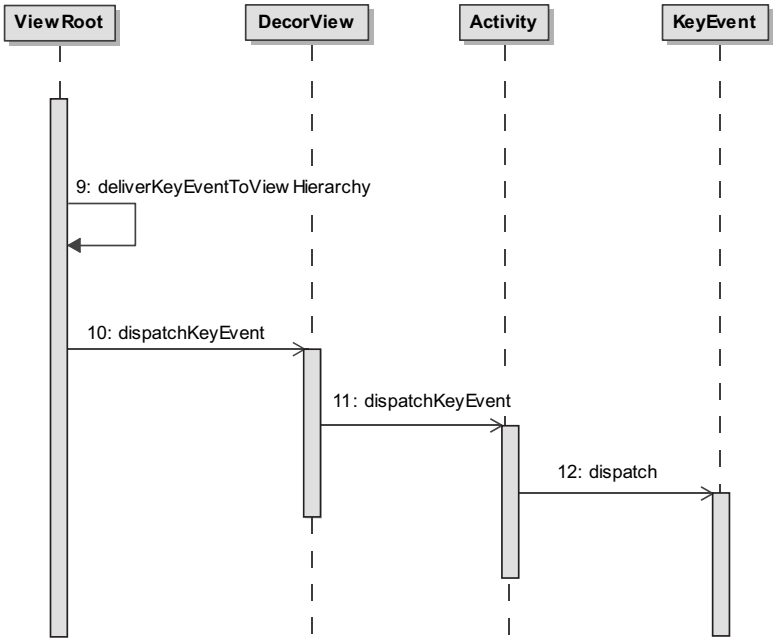


图14–16　系统当前激活的应用程序窗口获得键盘消息的过程——Step 9至Step 12

Step 9：ViewRoot.deliverKeyEventToViewHierarchy

frameworks/base/core/java/android/view/ViewRoot.java

```
01 public final class ViewRoot extends Handler implements ViewParent,
02         View.AttachInfo.Callbacks {
03     ......
04
05     private void deliverKeyEventToViewHierarchy(KeyEvent event, boolean sendDone) {
06         try {
07             if (mView != null && mAdded) {
08                 ......
09
10                 boolean keyHandled = mView.dispatchKeyEvent(event);
11             }
12             ......
13         } finally {
14             if (sendDone) {
15                 finishInputEvent();
16             }
17         }
18     }
19
20     ......
21 }
```

第7行的if语句首先检查当前正在处理的ViewRoot对象是否关联了一个Decorview对象。从前面14.3.1小节的Step 1可以知道，如果关联了，那么它的成员变量mView就会指向这个Decorview对象，并且它的成员变量mAdded的值会被设置为true。接下来第10行代码就会调用这个Decorview对象的成员函数dispatchKeyEvent来继续处理参数event所描述的一个键盘事件。

与当前正在处理的ViewRoot对象所关联的一个Decorview对象在处理完成参数event所描述的一个键盘事件之后，第14行的if语句就会判断参数sendDone的值是否等于true。如果等于true，那么接下来

第15行代码就会调用当前正在处理的ViewRoot对象的成员函数finishInputEvent向InputDispatcher发送一个键盘事件处理完成通知。

下面我们首先分析DecroView类的成员函数dispatchKeyEvent的实现，然后分析ViewRoot类的成员函数finishInputEvent的实现。

Step 10：DecorView.dispatchKeyEvent

frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindow.java

```
01 public class PhoneWindow extends Window implements MenuBuilder.Callback {
02     ......
03
04     private final class DecorView extends FrameLayout implements RootViewSurfaceTaker {
05         ......
06
07         /** The feature ID of the panel, or -1 if this is the application's DecorView */
08         private final int mFeatureId;
09         ......
10
11         @Override
12         public boolean dispatchKeyEvent(KeyEvent event) {
13             ......
14
15             final Callback cb = getCallback();
16             final boolean handled = cb != null && mFeatureId < 0 ? cb.dispatchKeyEvent(event)
17                 : super.dispatchKeyEvent(event);
18
19             ......
20         }
21
22         ......
23     }
24
25     ......
26 }
```

当一个DecorView对象是与一个应用程序窗口关联在一起的时候，它的成员变量mFeatureId的值就会等于-1。在这种情况下，第15行代码就会首先获得与它所关联的一个回调接口，接着再调用这个回调接口的成员函数dispatchKeyEvent来处理参数event所描述的一个键盘事件。

假设当前正在处理的DecorView对象是与一个应用程序窗口关联在一起的，即它的成员变量mFeatureId的值等于-1，接下来我们就继续分析如何获得与它所关联的一个回调接口。

第15行代码调用了外部类PhoneWindow的成员函数getCallback来获得与当前正在处理的DecorView对象所关联的一个回调接口。PhoneWindow类是从Window类继承下来的，它的成员函数getCallback也是从Window类继承下来的。因此，第15行代码实际上是调用Window类的成员函数getCallback来获得一个回调接口的。

从前面7.2小节的Step 34可以知道，系统在启动一个Activity组件时，会调用它的成员函数attach来执行一些初始化工作，其中的一个工作就是为这个Activity组件创建一个PhoneWindow对象，如下所示。

frameworks/base/core/java/android/app/Activity.java

```
01 public class Activity extends ContextThemeWrapper
02         implements LayoutInflater.Factory,
03         Window.Callback, KeyEvent.Callback,
04         OnCreateContextMenuListener, ComponentCallbacks {
05     ......
06
07     private Window mWindow;
```

```
08      ......
09
10      final void attach(Context context, ActivityThread aThread,
11              Instrumentation instr, IBinder token, int ident,
12              Application application, Intent intent, ActivityInfo info,
13              CharSequence title, Activity parent, String id,
14              Object lastNonConfigurationInstance,
15              HashMap<String,Object> lastNonConfigurationChildInstances,
16              Configuration config) {
17          ......
18
19          mWindow = PolicyManager.makeNewWindow(this);
20          mWindow.setCallback(this);
21          ......
22
23      }
24
25      ......
26  }
```

第19行代码首先调用PolicyManager类的静态成员函数makeNewWindow为当前正在启动的Activity组件创建一个PhoneWindow对象，并且保存在它的成员变量mWindow中，这样就可以将一个PhoneWindow对象与一个Activity组件关联起来。接着第20行代码又会将当前正在启动的Activity组件所实现的一个Window.Callback接口设置到与它所关联的一个PhoneWindow对象中，以便这个PhoneWindow对象可以通过这个接口将接收到的键盘事件分发给与它所关联的Activity组件处理。

从这里就可以看出，DecorView类的成员函数dispatchKeyEvent调用外部类PhoneWindow的成员函数getCallback所获得的一个回调接口实际上指向的是一个Activity组件，即指向的是系统当前激活的应用程序窗口。接下来就会调用它的成员函数dispatchKeyEvent，以便可以将参数event所描述的一个键盘事件分发给它处理。

Step 11：Activity.dispatchKeyEvent

frameworks/base/core/java/android/app/Activity.java

```
01 public class Activity extends ContextThemeWrapper
02          implements LayoutInflater.Factory,
03          Window.Callback, KeyEvent.Callback,
04          OnCreateContextMenuListener, ComponentCallbacks {
05      ......
06
07      private Window mWindow;
08      ......
09
10      /*package*/ View mDecor = null;
11      ......
12
13      public boolean dispatchKeyEvent(KeyEvent event) {
14          onUserInteraction();
15          Window win = getWindow();
16          if (win.superDispatchKeyEvent(event)) {
17              return true;
18          }
19          View decor = mDecor;
20          if (decor == null) decor = win.getDecorView();
21          return event.dispatch(this, decor != null
22                  ? decor.getKeyDispatcherState() : null, this);
23      }
24
```

```
25     ......
26 }
```

第14行代码首先调用Activity类的成员函数onUserInteraction来通知系统当前激活的应用程序窗口，用户通过键盘与它发生了一个交互操作；接着第16行代码再调用与系统当前激活的应用程序窗口所关联的一个PhoneWindow对象的成员函数superDispatchKeyEvent来处理参数event所描述的一个键盘事件。如果这个PhoneWindow对象的成员函数superDispatchKeyEvent的返回值等于true，那么参数event所描述的一个键盘事件的处理过程就结束了；否则，接下来第21行和第22行代码就会调用参数event所指向的一个KeyEvent对象的成员函数dispatch来继续分发它本身所描述的一个键盘事件。

Step 12：KeyEvent.dispatch

frameworks/base/core/java/android/view/KeyEvent.java

```
01 public class KeyEvent extends InputEvent implements Parcelable {
02     ......
03
04     public final boolean dispatch(Callback receiver, DispatcherState state,
05             Object target) {
06         switch (mAction) {
07         case ACTION_DOWN: {
08             ......
09             boolean res = receiver.onKeyDown(mKeyCode, this);
10             ......
11             return res;
12         }
13         case ACTION_UP:
14             ......
15             return receiver.onKeyUp(mKeyCode, this);
16         case ACTION_MULTIPLE:
17             final int count = mRepeatCount;
18             final int code = mKeyCode;
19             if (receiver.onKeyMultiple(code, count, this)) {
20                 return true;
21             }
22             ......
23         }
24         return false;
25     }
26
27     ......
28 }
```

KeyEvent类的成员变量mAction用来描述一个按键操作，其中，ACTION_DOWN和ACTION_UP分别用来描述一个按键被按下和被松开一次的操作；而ACTION_MULTIPLE用来描述一个键盘按键被重复地按下和松开的操作，重复的次数保存在成员变量mRepeatCount中。KeyEvent类的成员变量mKeyCode用来描述一个键盘码，即用来描述被按下或者被松开的是哪一个键盘按键。

参数receiver指向了一个类型为KeyEvent.Callback的回调接口，从前面的Step 11可以知道，这个回调接口实际上指向的是一个Activity组件，第6行到第23行的switch语句分别调用它的成员函数onKeyDown、onKeyUp和onKeyMultiple来处理一个键盘按键被按下、被松开，以及被重复地按下和松开的操作。从这里就可以看出，我们在自定义一个Activity组件时，如果需要处理键盘事件，那么就需要重写父类Activity的成员函数onKeyDown、onKeyUp和onKeyMultiple。

这一步执行完成之后，系统当前激活的应用程序窗口就处理完成一个键盘事件了。返回到前面的Step 9中，即ViewRoot类的成员函数deliverKeyEventToViewHierarchy中，接下来它就会调用ViewRoot

类的成员函数finishInputEvent向InputDispatcher发送一个键盘事件处理完成通知，如图14-17所示。

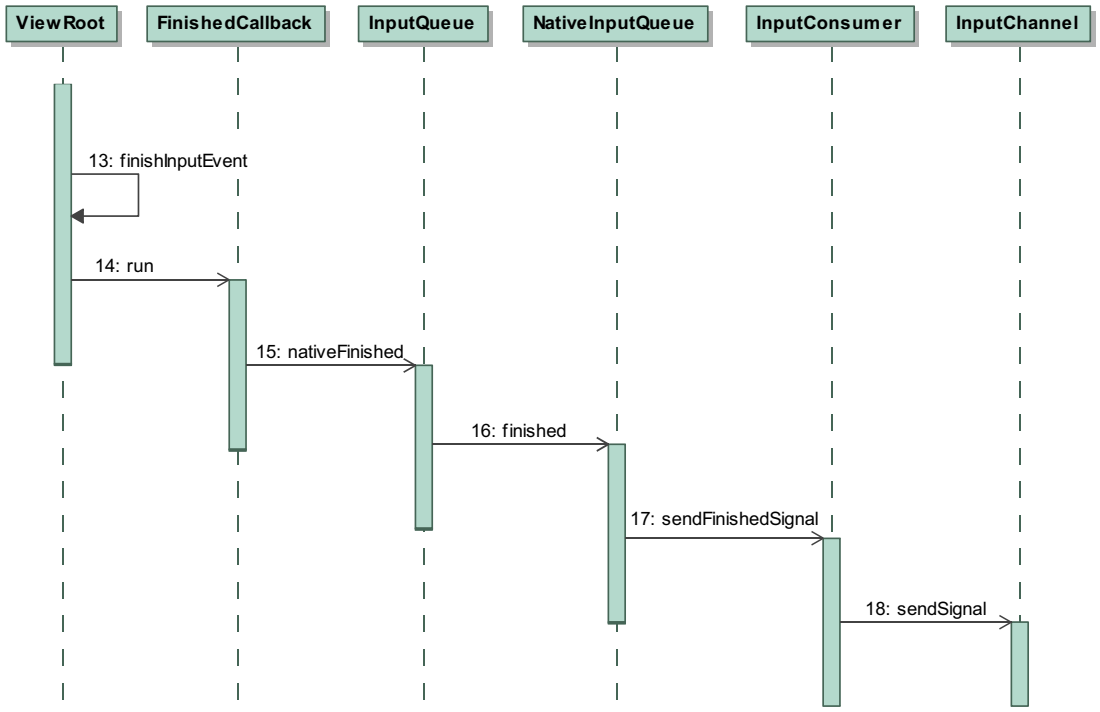


图14-17　系统当前激活的应用程序窗口获得键盘消息的过程——Step 13至Step 18

Step 13：ViewRoot.finishInputEvent

frameworks/base/core/java/android/view/ViewRoot.java

```
01 public final class ViewRoot extends Handler implements ViewParent,
02         View.AttachInfo.Callbacks {
03     ......
04
05     private void finishInputEvent() {
06         ......
07
08         if (mFinishedCallback != null) {
09             mFinishedCallback.run();
10             ......
11         }
12         ......
13     }
14     ......
15
16     private Runnable mFinishedCallback;
17     ......
18 }
```

从前面的Step 5和Step 6可以知道，ViewRoot类的成员变量mFinishedCallback指向了一个类型为FinishedCallback的回调接口，第9行代码调用它的成员函数run向InputDispatcher发送一个键盘事件处理完成通知。

Step 14：FinishedCallback.run

frameworks/base/core/java/android/view/InputQueue.java

```
01 public final class InputQueue {
02     ......
```

```
03
04      private static native void nativeFinished(long finishedToken);
05      ......
06
07      private static class FinishedCallback implements Runnable {
08          ......
09
10          private long mFinishedToken;
11          ......
12
13          public void run() {
14              synchronized (sLock) {
15                  ......
16
17                  nativeFinished(mFinishedToken);
18                  ......
19              }
20          }
21
22          ......
23      }
24
25      ......
26  }
```

FinishedCallback类的成员变量mFinishedToken是一个长整数，它用来描述前面所处理的一个键盘事件的令牌。从前面的Step 1可以知道，通过这个令牌就可以找到一个Connection对象，然后通过这个Connection对象就可以向InputDispatcher发送一个键盘事件处理完成通知。

第17行代码以FinishedCallback类的成员变量mFinishedToken为参数来调用外部类InputQueue的静态成员函数nativeFinished，以便可以向InputDispatcher发送一个键盘事件处理完成通知。

Step 15：InputQueue.nativeFinished

frameworks/base/core/jni/android_view_InputQueue.cpp

```
1 static void android_view_InputQueue_nativeFinished(JNIEnv* env, jclass clazz,
2          jlong finishedToken) {
3      status_t status = gNativeInputQueue.finished(
4          env, finishedToken, false /*ignoreSpuriousFinish*/);
5
6      ......
7  }
```

InputQueue类的静态成员函数nativeFinished是一个JNI方法，它是由C++层的函数android_view_InputQueue_nativeFinished来实现的。

从前面14.3.4小节的Step 2可以知道，gNativeInputQueue是一个全局变量，它指向了一个NativeInputQueue对象，第3行和第4行代码调用它的成员函数finished向InputDispatcher发送一个键盘事件处理完成通知。

Step 16：NativeInputQueue.finished

frameworks/base/core/jni/android_view_InputQueue.cpp

```
01 status_t NativeInputQueue::finished(JNIEnv* env, jlong finishedToken, bool ignoreSpuriousFinish) {
02      int32_t receiveFd;
03      uint16_t connectionId;
04      uint16_t messageSeqNum;
05      parseFinishedToken(finishedToken, &receiveFd, &connectionId, &messageSeqNum);
06
07      { // acquire lock
08          AutoMutex _l(mLock);
09
10          ssize_t connectionIndex = mConnectionsByReceiveFd.indexOfKey(receiveFd);
11          ......
12
```

```
13          sp<Connection> connection = mConnectionsByReceiveFd.valueAt(connectionIndex);
14          ......
15
16          status_t status = connection->inputConsumer.sendFinishedSignal();
17          ......
18      } // release lock
19
20      return OK;
21  }
```

第5行代码首先调用NativeInputQueue类的成员函数parseFinishedToken来解析参数finishedToken所描述的一个键盘事件令牌，如下所示。

frameworks/base/core/jni/android_view_InputQueue.cpp

```
1 void NativeInputQueue::parseFinishedToken(jlong finishedToken,
2          int32_t* outReceiveFd, uint16_t* outConnectionId, uint16_t* outMessageIndex) {
3      *outReceiveFd = int32_t(finishedToken >> 32);
4      *outConnectionId = uint16_t(finishedToken >> 16);
5      *outMessageIndex = uint16_t(finishedToken);
6 }
```

从前面的Step 1可以知道，参数finishedToken的高32位、中间16位和低16位分别保存着一个前向管道的读端文件描述符，以及一个Connection对象的成员变量id和messageSeqNum的值，第3行到第5行代码将它们取出来，并且分别保存在输出参数outReceiveFd、outConnectionId和outMessageIndex中。

回到NativeInputQueue类的成员函数finished中，这时候变量receiveFd就用来描述一个前向管道的读端文件描述符；接着第10行和第13行代码再根据这个文件描述符在NativeInputQueue类的成员变量mConnectionsByReceiveFd中找到一个对应的Connection对象；最后第16行代码再调用这个Connection对象的成员变量inputConsumer所指向的一个InputConsumer对象的成员函数sendFinishedSignal向InputDispatcher发送一个键盘事件处理完成通知。

Step 17：InputConsumer.sendFinishedSignal

frameworks/base/libs/ui/InputTransport.cpp

```
1 status_t InputConsumer::sendFinishedSignal() {
2      ......
3
4      return mChannel->sendSignal(INPUT_SIGNAL_FINISHED);
5 }
```

InputConsumer类的成员变量mChannel指向了一个Client端InputChannel，第4行代码调用它的成员函数sendSignal向InputDispatcher发送一个键盘事件处理完成通知。

Step 18：InputChannel.sendSignal

frameworks/base/libs/ui/InputTransport.cpp

```
01 status_t InputChannel::sendSignal(char signal) {
02      ssize_t nWrite;
03      do {
04          nWrite = ::write(mSendPipeFd, & signal, 1);
05      } while (nWrite == -1 && errno == EINTR);
06
07      if (nWrite == 1) {
08          ......
09          return OK;
10      }
11
12      return -errno;
13 }
```

从前面14.3.1小节的Step 6可以知道，Client端InputChannel的成员变量mSendPipeFd用来描述一个

反向管道的写端文件描述符，第3行到第5行代码通过这个文件描述符向这个反向管道写入一个字符signal（INPUT_SIGNAL_FINISHED），以便可以向InputDispatcher发送一个键盘事件处理完成通知。

从前面14.3.2小节的Step 4可以知道，当系统当前激活的应用程序窗口向InputDispatcher发送了一个键盘事件处理完成通知时，InputDispatcher就会被唤醒，以便可以继续将下一个键盘事件分发给系统当前激活的应用程序窗口处理。

接下来，我们就继续分析InputDispatcher获得一个键盘事件处理完成通知的过程。

14.4.4　InputDispatcher获得键盘事件处理完成通知

从前面14.3.2小节的Step 4可以知道，当系统当前激活的应用程序窗口向InputDispatcher发送了一个键盘事件处理完成通知时，InputDispatcher就会调用InputDispatcher类的静态成员函数handleReceiveCallback来处理这个通知，如图14-18所示。

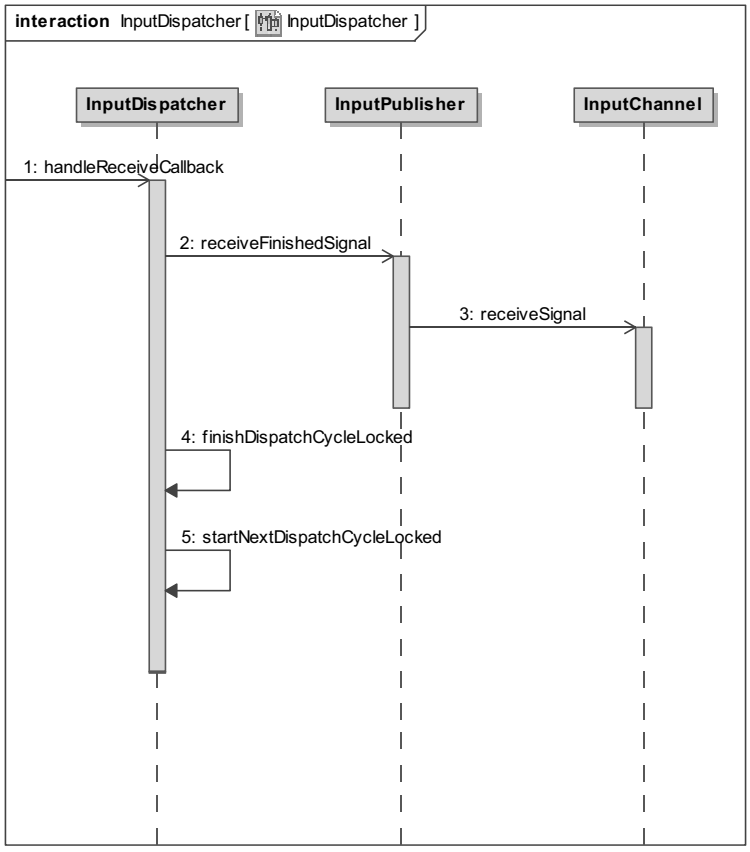


图14-18　InputDispatcher获得键盘事件处理完成通知的过程

这个过程一共分为5个步骤，下面我们就详细分析每一个步骤。

Step 1：InputDispatcher.handleReceiveCallback

frameworks/base/libs/ui/InputDispatcher.cpp

```
01 int InputDispatcher::handleReceiveCallback(int receiveFd, int events, void* data) {
02     InputDispatcher* d = static_cast<InputDispatcher*>(data);
03
04     { // acquire lock
05         AutoMutex _l(d->mLock);
06
07         ssize_t connectionIndex = d->mConnectionsByReceiveFd.indexOfKey(receiveFd);
```

```
08          ......
09
10          nsecs_t currentTime = now();
11
12          sp<Connection> connection = d->mConnectionsByReceiveFd.valueAt(connectionIndex);
13          ......
14
15          status_t status = connection->inputPublisher.receiveFinishedSignal();
16          ......
17
18          d->finishDispatchCycleLocked(currentTime, connection);
19          ......
20
21          return 1;
22      } // release lock
23  }
```

从前面14.3.2小节的Step 4可以知道，参数data指向了一个InputDispatcher对象，因此，第2行代码就可以安全地将它转换为一个InputDispatcher对象，并且保存在变量d中。

从前面14.3.2小节的Step 4还可以知道，参数receiveFd是用来描述一个Server端InputChannel内部的一个反向管道的读端文件描述符的，第7行和第12行代码通过它在InputDispatcher对象d的成员变量mConnectionsByReceiveFd中找到一个对应的Connection对象connection。这个Connection对象是用来在InputDisptacher和系统当前激活的应用程序窗口之间建立连接的。有了这个Connection对象之后，InputDispatcher就可以接收到系统当前激活的应用程序窗口发送过来的键盘事件处理完成通知了。

InputDispatcher接收到系统当前激活的应用程序窗口发送过来的键盘事件处理完成通知之后，就可以准备向系统当前激活的应用程序窗口分发下一个键盘事件了，这是通过调用InputDispatcher的成员函数finishDispatchCycleLocked来实现的，如第18行代码所示。不过，InputDispatcher在准备分发下一个键盘事件之前，首先要调用前面所获得的一个Connection对象的成员变量inputPublisher所指向的一个InputPublisher对象的成员函数receiveFinishedSignal，来确认系统当前激活的应用程序窗口的确是向InputDispatcher发送了一个键盘事件处理完成通知，如第15行代码所示。

接下来，我们首先分析InputPublisher类的成员函数receiveFinishedSignal的实现，以便可以了解它是如何确认系统当前激活的应用程序窗口发送了一个键盘事件处理完成通知给InputDisptacher的；然后分析InputDispatcher类的成员函数finishDispatchCycleLocked的实现，以便可以了解它是如何准备向系统当前激活的应用程序窗口分发下一个键盘事件的。

Step 2：InputPublisher.receiveFinishedSignal

frameworks/base/libs/ui/InputTransport.cpp

```
01 status_t InputPublisher::receiveFinishedSignal() {
02      ....
03
04      char signal;
05      status_t result = mChannel->receiveSignal(& signal);
06      if (result) {
07          return result;
08      }
09      if (signal != INPUT_SIGNAL_FINISHED) {
10          .......
11          return UNKNOWN_ERROR;
12      }
13      return OK;
14  }
```

InputPublisher类的成员变量mChannel指向了一个Server端InputChannel，第5行代码首先调用它的

成员函数receiveSignal从它内部的一个反向管道中读出一个字符，接着第9行的if语句再检查这个字符的值是否等于INPUT_SIGNAL_FINISHED。从前面14.4.3小节的Step 18可以知道，系统当前激活的应用程序窗口向这个反向管道写入的字符的值正好等于INPUT_SIGNAL_FINISHED，因此，接下来第13行代码就会返回一个OK值给调用者，表示系统当前激活的应用程序窗口确实是向InputDispatcher发送了一个键盘事件处理完成通知。

接下来，我们继续分析Server端InputChannel类的成员函数receiveSignal是如何从一个反向管道中读出一个字符的。

Step 3：InputChannel. receiveSignal

frameworks/base/libs/ui/InputTransport.cpp

```
01 status_t InputChannel::receiveSignal(char* outSignal) {
02     ssize_t nRead;
03     do {
04         nRead = ::read(mReceivePipeFd, outSignal, 1);
05     } while (nRead == -1 && errno == EINTR);
06
07     if (nRead == 1) {
08         ......
09         return OK;
10     }
11
12     ......
13     return -errno;
14 }
```

从前面14.3.1小节的Step 6可以知道，Server端InputChannel的成员变量mReceivePipeFd是用来描述一个反向管道的读端文件描述符的。因此，第3行到第5行代码就可以通过这个文件描述符从它所描述的一个反向管道中读出一个字符值，并且保存在输出参数outSignal中。

这一步执行完成之后，返回到前面的Step 1中，即InputDispatcher类的静态成员函数handleReceiveCallback中，接下来就会继续调用InputDispatcher类的成员函数finishDispatchCycleLocked来准备分发下一个键盘事件给系统当前激活的应用程序窗口处理。

Step 4：InputDispatcher.finishDispatchCycleLocked

frameworks/base/libs/ui/InputDispatcher.cpp

```
01 void InputDispatcher::finishDispatchCycleLocked(nsecs_t currentTime,
02         const sp<Connection>& connection) {
03     ......
04
05     // Notify other system components.
06     onDispatchCycleFinishedLocked(currentTime, connection);
07
08     // Reset the publisher since the event has been consumed.
09     // We do this now so that the publisher can release some of its internal resources
10     // while waiting for the next dispatch cycle to begin.
11     status_t status = connection->inputPublisher.reset();
12     ......
13
14     startNextDispatchCycleLocked(currentTime, connection);
15 }
```

第6行代码首先调用InputDispatcher类的成员函数onDispatchCycleFinishedLocked来通知其他的系统组件，系统当前激活的应用程序窗口已经处理完成一个键盘事件了；然后第11行代码调用参数connection所描述的一个Connection对象的成员变量inputPublisher所指向的一个InputPublisher对象的成员函数reset，来重置它内部引用的一个匿名共享内块，以便接下来可以将下一个需要分发的键盘事件的数据写入到它里面去；最后第14行代码调用InputDispatcher类的成员函数startNextDispatchCycleLocked来

准备将下一个键盘事件分发给系统当前激活的应用程序窗口处理。

Step 5：InputDispatcher. startNextDispatchCycleLocked

frameworks/base/libs/ui/InputDispatcher.cpp

```
01 void InputDispatcher::startNextDispatchCycleLocked(nsecs_t currentTime,
02         const sp<Connection>& connection) {
03     // Start the next dispatch cycle for this connection.
04     while (! connection->outboundQueue.isEmpty()) {
05         DispatchEntry* dispatchEntry = connection->outboundQueue.headSentinel.next;
06         if (dispatchEntry->inProgress) {
07             ......
08
09             // Finished.
10             connection->outboundQueue.dequeueAtHead();
11             ......
12
13             mAllocator.releaseDispatchEntry(dispatchEntry);
14         } else {
15             // If the head is not in progress, then we must have already dequeued the in
16             // progress event, which means we actually aborted it.
17             // So just start the next event for this connection.
18             startDispatchCycleLocked(currentTime, connection);
19             return;
20         }
21     }
22
23     ......
24 }
```

在前面14.4.2小节的Step 5中提到，每一个Connection对象的内部都有一个独立的队列outboundQueue，用来保存那些正在等待分发给系统当前激活的应用程序窗口处理的键盘事件。现在既然系统当前激活的应用程序窗口已经处理完成InputDispatcher上一次分发给它的键盘事件了，InputDisptacher接下来就可以继续将保存在这个队列中的下一个键盘事件分发给系统当前激活的应用程序窗口处理了。

第4行到第21行的while循环首先在参数connection所描述的一个Connection对象的内部的一个待分发键盘事件队列中找到上一次已经分发给系统当前激活的应用程序窗口处理的键盘事件，以便可以将这个键盘事件从队列中删除，以及将它所占用的内存释放，如第10行和第13行代码所示。

从前面14.4.2小节的Step 6可以知道，InputDispatcher在向系统当前激活的应用程序窗口分发一个键盘事件之前，会将用来描述该键盘事件的一个DispatchEntry结构体的成员变量inProgress的值设置为true。因此，通过检查一个DispatchEntry结构体的成员变量inProgress的值是否等于true，就可以知道它所描述的一个键盘事件是否已经分发给系统当前激活的应用程序窗口处理了，如第6行的if语句所示。

第4行到第21行的while循环接着在参数connection所描述的一个Connection对象的内部的一个待分发键盘事件队列中找到下一个待分发的键盘事件，最后就调用InputDispatcher类的成员函数startDispatchCycleLocked将它分发给系统当前激活的应用程序窗口处理。这个过程可以参考前面14.4.2小节中的Step 6。

至此，一个键盘消息的完整分发过程就分析完成了。接下来我们继续分析系统当前激活的应用程序窗口与输入管理器InputManager断开连接的过程，即InputChannel的注销过程。

14.5 InputChannel的注销过程

一个应用程序窗口在创建的过程中，会与系统输入管理器InputManager建立一个交叉连接，以便可以接收到系统的键盘事件。从前面14.3小节的内容可以知道，这个交叉连接是由一个Server端Input

Channel和一个Client端InputChannel来描述的，它们分别注册在系统输入管理器InputManager，以及应用程序窗口所运行在的应用程序进程的主线程中。

与创建过程相反，一个应用程序窗口在销毁的过程中，会断开与系统输入管理器InputManager之间的交叉连接，即将前面所注册的一个Server端InputChannel和一个Client端InputChannel注销，避免以后再接收到系统的键盘事件。

由于一个Client端InputChannel和一个Server端InputChannel是在它们所关联的一个应用程序窗口的销毁过程中被注销的，因此，接下来我们首先分析应用程序窗口的销毁过程，然后再分析Client端InputChannel和Server端InputChannel的注销过程。

14.5.1　销毁应用程序窗口

一个Activity组件的销毁过程，即一个应用程序窗口的销毁过程，是由Activity管理服务ActivityManagerService发起的。每次启动了一个新的Activity组件之后，Activity管理服务ActivityManagerService都会向它所运行在的线程的消息队列发送一个类型为IDLE_NOW_MSG的消息。这个消息最终是由ActivityStack类的成员函数activityIdleInternal来处理的，它所要完成的工作便是将那些不再需要的Activity组件销毁掉。

Activity管理服务ActivityManagerService在销毁一个Activity组件之前，会向相应的应用程序进程发送一个Activity组件销毁通知，以便它可以执行一些清理工作。一个应用程序进程接收到一个Activity组件销毁通知之后，就会注销与这个Activity组件所关联的一个Client端InputChannel和一个Server端InputChannel，以便可以断开这个Activity组件与系统输入管理器InputManager之间的交叉连接。

接下来，我们就从ActivityStack类的成员函数activityIdleInternal开始分析一个应用程序窗口的销毁过程。这个过程一共分为11个步骤，下面我们就详细分析每一个步骤。

第1步到第4步如图14-19所示。

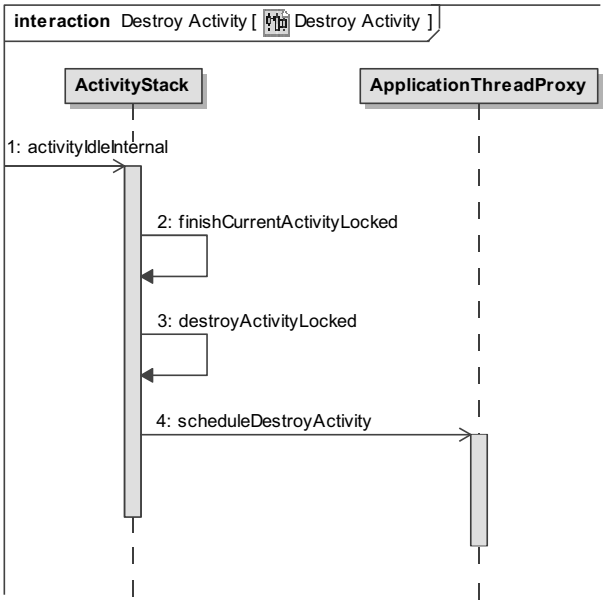


图14-19　应用程序窗口的销毁过程——Step 1至Step 4

Step 1：ActivityStack.activityIdleInternal

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02     ......
03
04     final void activityIdleInternal(IBinder token, boolean fromTimeout,
05             Configuration config) {
06         ......
07
08         ArrayList<ActivityRecord> stops = null;
09         ......
10
11         int NS = 0;
12         ......
13
14         synchronized (mService) {
15             ......
16
17             // Atomically retrieve all of the other things to do.
18             stops = processStoppingActivitiesLocked(true);
19             NS = stops != null ? stops.size() : 0;
20             ......
21         }
22
23         int i;
24         ......
25
26         // Stop any activities that are scheduled to do so but have been
27         // waiting for the next one to start.
28         for (i=0; i<NS; i++) {
29             ActivityRecord r = (ActivityRecord)stops.get(i);
30             synchronized (mService) {
31                 if (r.finishing) {
32                     finishCurrentActivityLocked(r, FINISH_IMMEDIATELY);
33                 }
34                 ......
35             }
36         }
37
38         ......
39     }
40
41     ......
42 }
```

在ActivityStack类中，每一个Activity组件都是使用一个ActivityRecord对象来描述的。

第18行代码首先调用ActivityStack类的成员函数processStoppingActivitiesLocked来收集那些可能需要销毁的Activity组件，并且将它们保存在一个ActivityRecord列表stops中。

ActivityStack类的成员函数processStoppingActivitiesLocked的实现如下所示。

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02     ......
03
04     /**
05      * List of activities that are ready to be stopped, but waiting
06      * for the next activity to settle down before doing so.  It contains
```

```
07      * HistoryRecord objects.
08      */
09     final ArrayList<ActivityRecord> mStoppingActivities
10             = new ArrayList<ActivityRecord>();
11     ......
12
13     final ArrayList<ActivityRecord> processStoppingActivitiesLocked(
14             boolean remove) {
15         int N = mStoppingActivities.size();
16         if (N <= 0) return null;
17
18         ArrayList<ActivityRecord> stops = null;
19         ......
20
21         for (int i=0; i<N; i++) {
22             ActivityRecord s = mStoppingActivities.get(i);
23             ......
24
25             if (!s.waitingVisible && remove) {
26                 ......
27                 if (stops == null) {
28                     stops = new ArrayList<ActivityRecord>();
29                 }
30                 stops.add(s);
31                 mStoppingActivities.remove(i);
32                 N--;
33                 i--;
34             }
35         }
36
37         return stops;
38     }
39
40     ......
41 }
```

从前面的调用过程可以知道，参数remove的值等于true，表示要将那些处于Stopped状态的Activity组件从系统中移除。

ActivityStack类的成员变量mStoppingActivities保存了那些处于Stopped状态的Activity组件，第21行到第35行的for循环依次遍历这些Activity组件，以便可以将它们返回给调用者来销毁。

如果一个处于Stopped状态的Activity组件不是处于被重新启动的状态，即用来描述它的一个ActivityRecord对象的成员变量waitingVisible的值等于false，那么这时候就可能需要将它销毁。因此，第30行代码就会将它添加到一个ActivityRecord列表stops中，以便可以返回给调用者。

回到ActivityStack类的成员函数activityIdleInternal中，这时候可能需要销毁的Activity组件就保存在ActivityRecord列表stops中了，接下来第28行到第36行的for循环依次检查这些Activity组件是否真的需要销毁。如果用来描述一个处于Stopped状态的Activity组件的ActivityRecord对象的成员变量finishing的值等于true，那么就说明这个Activity组件是需要销毁的。因此，第32行代码就会调用ActivityStack类的成员函数finishCurrentActivityLocked来对它进行销毁。

Step 2：ActivityStack.finishCurrentActivityLocked

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02     ......
03
```

```
04     private final ActivityRecord finishCurrentActivityLocked(ActivityRecord r,
05             int mode) {
06         final int index = indexOfTokenLocked(r);
07         if (index < 0) {
08             return null;
09         }
10
11         return finishCurrentActivityLocked(r, index, mode);
12     }
13
14     ......
15 }
```

第6行代码首先调用ActivityStack类的成员函数indexOfTokenLocked来得到即将销毁的一个Activity组件在系统的Activity组件堆栈中的位置，接着第11行代码调用另外一个重载版本的成员函数finishCurrentActivityLocked来销毁该Activity组件。

另外一个重载版本的成员函数finishCurrentActivityLocked的实现如下所示。

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02     ......
03
04     private final ActivityRecord finishCurrentActivityLocked(ActivityRecord r,
05             int index, int mode) {
06         ......
07
08         final ActivityState prevState = r.state;
09         r.state = ActivityState.FINISHING;
10
11         if (mode == FINISH_IMMEDIATELY
12             || prevState == ActivityState.STOPPED
13             || prevState == ActivityState.INITIALIZING) {
14             // If this activity is already stopped, we can just finish
15             // it right now.
16             return destroyActivityLocked(r, true) ? null : r;
17         }
18         ......
19
20         return r;
21     }
22
23     ......
24 }
```

第9行代码首先将参数r所描述的一个Activity组件的状态设置为ActivityState.FINISHING，以便可以表示这个Activity组件正在被销毁。从前面的调用过程可以知道，参数mode的值等于FINISH_IMMEDIATELY，因此，接下来第16行代码就会调用ActivityStack类的成员函数destroyActivityLocked来销毁参数r所描述的一个Activity组件。

Step 3：ActivityStack.destroyActivityLocked

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02     ......
03
04     final boolean destroyActivityLocked(ActivityRecord r,
05             boolean removeFromApp) {
06         ......
07
```

```
08          if (hadApp) {
09              ......
10
11              try {
12                  ......
13                  r.app.thread.scheduleDestroyActivity(r, r.finishing,
14                          r.configChangeFlags);
15              } catch (Exception e) {
16                  ......
17              }
18
19              ......
20          }
21
22          ......
23      }
24
25      ......
26  }
```

第13行和第14行代码首先获得参数r所描述的一个Activity组件所运行在的一个应用程序进程。这个应用程序进程是使用一个ProcessRecord对象来描述的。接着再通过这个ProcessRecord对象的成员变量thread来获得一个类型为ApplicationThreadProxy的Binder代理对象。有了这个Binder代理对象之后，最后就可以调用它的成员函数scheduleDestroyActivity向参数r所描述的一个Activity组件所运行在的一个应用程序进程发送一个Activity组件销毁通知。

Step 4：ApplicationThreadProxy. scheduleDestroyActivity

frameworks/base/core/java/android/app/ApplicationThreadNative.java

```
01 class ApplicationThreadProxy implements IApplicationThread {
02      ......
03
04      public final void scheduleDestroyActivity(IBinder token, boolean finishing,
05              int configChanges) throws RemoteException {
06          Parcel data = Parcel.obtain();
07          data.writeInterfaceToken(IApplicationThread.descriptor);
08          data.writeStrongBinder(token);
09          data.writeInt(finishing ? 1 : 0);
10          data.writeInt(configChanges);
11          mRemote.transact(SCHEDULE_FINISH_ACTIVITY_TRANSACTION, data, null,
12                  IBinder.FLAG_ONEWAY);
13          data.recycle();
14      }
15
16      ......
17  }
```

第7行到第10行代码将从前面传进来的参数写入到Parcel对象data中，然后再通过ApplicationThreadProxy类内部的一个Binder代理对象mRemote，向参数token所描述的一个Activity组件所运行在的一个应用程序进程发送一个类型为SCHEDULE_FINISH_ACTIVITY_TRANSACTION的进程间通信请求。

以上4步都是在ActivityManagerService中执行的，接下来第5步到第11步是在即将要销毁的Activity组件所运行在的应用程序进程中执行的，主要是用来处理ActivityManagerService发出的类型为SCHEDULE_FINISH_ACTIVITY_TRANSACTION的进程间通信请求，如图14–20所示。

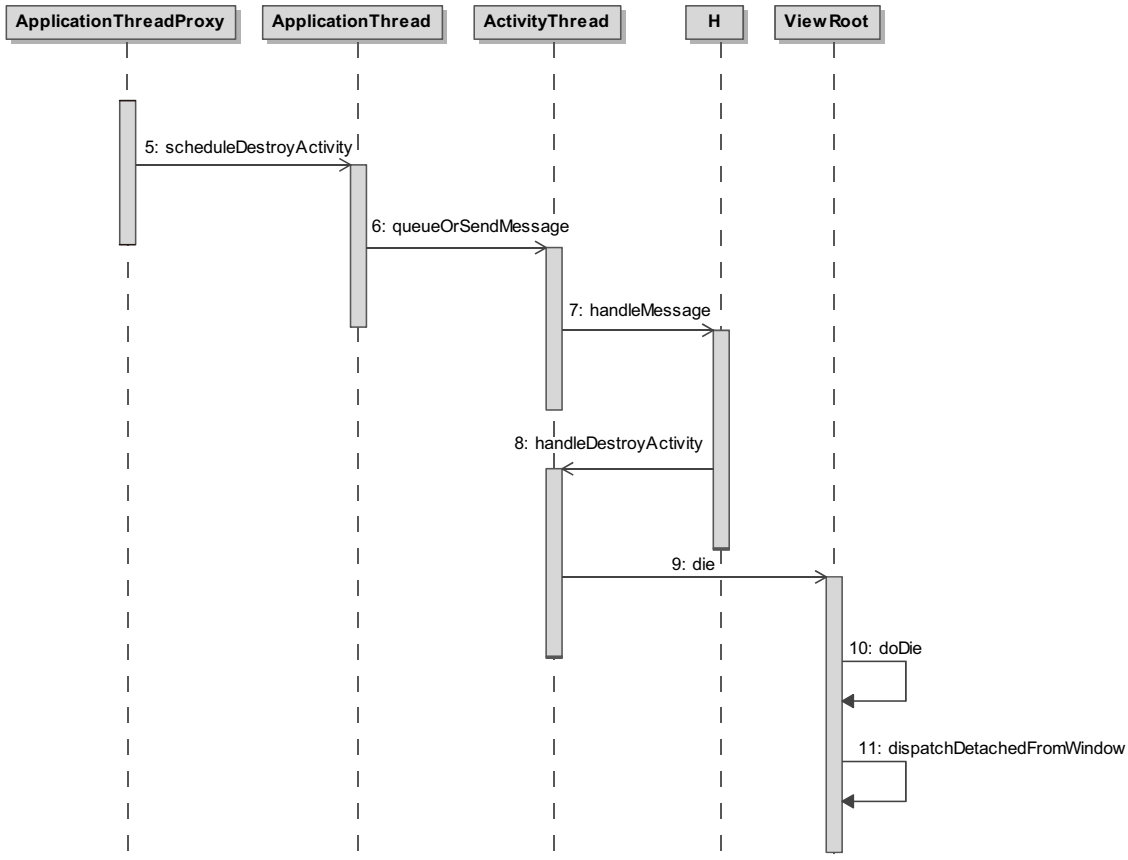


图14–20　应用程序窗口的销毁过程——Step 5至Step 11

Step 5：ApplicationThread.scheduleDestroyActivity

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final class ApplicationThread extends ApplicationThreadNative {
05         ......
06
07         public final void scheduleDestroyActivity(IBinder token, boolean finishing,
08                 int configChanges) {
09
10             queueOrSendMessage(H.DESTROY_ACTIVITY, token, finishing ? 1 : 0,
11                     configChanges);
12         }
13
14         ......
15     }
16
17     ......
18 }
```

ApplicationThread类的成员函数scheduleDestroyActivity用来处理类型为SCHEDULE_FINISH_ACTIVITY_TRANSACTION的进程间通信请求。

第10行和第11行代码调用了ActivityThread类的成员函数queueOrSendMessage，来向即将要销毁的Activity组件所运行在的应用程序进程的主线程的消息队列发送一个类型为DESTROY_ACTIVITY的消息。

Step 6：ActivityThread.queueOrSendMessage

第6步可以参考7.2小节中的第31步，它主要是将上一步即将要销毁的一个Activity组件封装成一个类型为DESTROY_ACTIVITY的消息，然后调用ActivityThread类的成员变量mH的成员函数sendMessage，将这个消息发送到即将要销毁的一个Activity组件所运行在的应用程序进程的主线程的消息队列中。这个消息最终是在ActivityThread类的成员变量mH的成员函数handleMessage中处理的。

Step 7：H.handleMessage

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final class H extends Handler {
05         ......
06
07         public void handleMessage(Message msg) {
08             ......
09
10             switch (msg.what) {
11                 ......
12                 case DESTROY_ACTIVITY:
13                     handleDestroyActivity((IBinder)msg.obj, msg.arg1 != 0,
14                             msg.arg2, false);
15                     break;
16                 ......
17             }
18
19             ......
20         }
21
22         ......
23     }
24
25     ......
26 }
```

第13行和第14行代码首先将参数msg的成员变量obj转换为一个IBinder接口，这个IBinder接口描述了一个即将要销毁的Activity组件；然后再调用ActivityThread类的成员函数handleDestroyActivity来对它进行销毁。

Step 8：ActivityThread.handleDestroyActivity

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     private final void handleDestroyActivity(IBinder token, boolean finishing,
05             int configChanges, boolean getNonConfigInstance) {
06         ......
07
08         ActivityClientRecord r = performDestroyActivity(token, finishing,
09             configChanges, getNonConfigInstance);
10         if (r != null) {
11             WindowManager wm = r.activity.getWindowManager();
12             View v = r.activity.mDecor;
13             if (v != null) {
14                 ......
15
16                 if (r.activity.mWindowAdded) {
```

```
17                      wm.removeViewImmediate(v);
18                  }
19                  ......
20              }
21              ......
22          }
23
24          ......
25      }
26
27      ......
28  }
```

第8行和第9行代码首先调用ActivityThread类的成员函数performDestroyActivity来通知参数token所描述的一个Activity组件，它要被销毁了，这会导致它的成员函数onDestroy被调用。ActivityThread类的成员函数performDestroyActivity执行完成之后，就会根据参数token的值返回一个ActivityClientRecord对象r，用来描述正在销毁的一个Activity组件。

ActivityClientRecord对象r的成员变量activity指向了一个正在销毁的Activity组件，第11行代码首先调用这个Activity组件的成员函数getWindowManager来获得一个应用程序窗口管理器wm，接着第12行代码通过这个Activity组件的成员变量mDecor来获得与它所关联的一个DecorView对象v。从前面14.3小节的内容可以知道，一个DecorView对象是用来描述一个应用程序窗口的，因此，DecorView对象v所描述的应用程序窗口即为与正在销毁的Activity组件所关联的一个应用程序窗口。

如果正在销毁的Activity组件关联了一个应用程序窗口，并且这个应用程序窗口也关联了一个ViewRoot对象，即第13行和第16行的两个if语句均为true，那么接下来第17行代码就会调用应用程序窗口管理器wm的成员函数removeViewImmediate从系统中移除与正在销毁的Activity组件所关联的一个应用程序窗口。

应用程序窗口管理器wm在移除DecorView对象v所描述的一个应用程序窗口的过程中，会找到与它所关联的一个ViewRoot对象，然后再调用这个ViewRoot对象的成员函数die来通知它正在被销毁。

Step 9：ViewRoot.die

frameworks/base/core/java/android/view/ViewRoot.java

```
01 public final class ViewRoot extends Handler implements ViewParent,
02         View.AttachInfo.Callbacks {
03     ......
04
05     public void die(boolean immediate) {
06         if (immediate) {
07             doDie();
08         } else {
09             sendEmptyMessage(DIE);
10         }
11     }
12
13     ......
14 }
```

参数immediate用来描述是否需要马上销毁一个ViewRoot对象。如果它的值等于true，那么第7行代码就会直接调用ViewRoot类的成员函数doDie来执行销毁操作；否则，第9行代码就会向正在销毁的Activity组件所运行在的应用程序进程的主线程的消息队列发送一个类型为DIE的消息。等到这个消息被处理时，正在销毁的Activity组件所运行在的应用程序进程的主线程再调用ViewRoot类的成员函数doDie来销毁与这个Activity组件所关联的一个ViewRoot对象。

Step 10：ViewRoot.doDie

frameworks/base/core/java/android/view/ViewRoot.java

```
01 public final class ViewRoot extends Handler implements ViewParent,
02         View.AttachInfo.Callbacks {
03     ......
04
05     void doDie() {
06         ......
07
08         synchronized (this) {
09             ......
10
11             if (mAdded) {
12                 mAdded = false;
13                 dispatchDetachedFromWindow();
14             }
15         }
16     }
17
18     ......
19 }
```

从前面14.3.1小节的Step 1可以知道，当我们将一个DecorView对象设置到一个ViewRoot对象里面时，除了会将这个ViewRoot对象的成员变量mAdded的值设置为true之外，还会为与这个ViewRoot对象所关联的一个应用程序窗口创建一个Client端InputChannel和一个Server端InputChannel，并且分别将它们注册到这个应用程序窗口所运行在的应用程序进程的主线程，以及系统输入管理器InputManager中。现在既然这个ViewRoot对象要被销毁了，接下来第13行代码就会调用ViewRoot类的成员函数dispatchDetachedFromWindow来注销前面为它所注册的一个Client端InputChannel和一个Server端InputChannel。

Step 11：ViewRoot.dispatchDetachedFromWindow

frameworks/base/core/java/android/view/ViewRoot.java

```
01 public final class ViewRoot extends Handler implements ViewParent,
02         View.AttachInfo.Callbacks {
03     ......
04
05     void dispatchDetachedFromWindow() {
06         ......
07
08         if (mInputChannel != null) {
09             if (mInputQueueCallback != null) {
10                 ......
11             } else {
12                 InputQueue.unregisterInputChannel(mInputChannel);
13             }
14         }
15
16         try {
17             sWindowSession.remove(mWindow);
18         } catch (RemoteException e) {
19         }
20
21         ......
22     }
23
24     ......
25 }
```

从前面的14.3.1和14.3.4小节可以知道，ViewRoot类的成员变量mInputChannel是用来描述一个Client端InputChannel的，并且这个Client端InputChannel被注册到了一个应用程序进程的主线程中。在前面14.3.1小节的Step 1中，我们假设ViewRoot类的成员变量mInputQueueCallback的值等于null，因此，接下来第12行代码就会调用InputQueue类的静态成员函数unregisterInputChannel来注销ViewRoot类的成员变量mInputChannel所描述的一个Client端InputChannel。

在前面14.3.1小节的Step 1中还提到，ViewRoot类的静态成员变量sWindowSession是一个Binder代理对象，它引用了运行在Window管理服务WindowManagerService中的一个类型为Session的Binder本地对象。第17行代码调用它的成员函数remove来请求Window管理服务WindowManagerService移除由ViewRoot类的成员变量mWindow所描述的一个应用程序窗口。在移除这个应用程序窗口的过程中，Window管理服务WindowManagerService还会注销与这个应用程序窗口所关联的一个Server端InputChannel。

至此，一个应用程序窗口的销毁过程就分析完成了。接下来我们继续分析与这个应用程序窗口所关联的一个Client端InputChannel和一个Server端InputChannel的注销过程。

14.5.2 注销Client端InputChannel

从前面14.5.1小节的Step 11可以知道，与一个应用程序窗口所关联的一个Client端InputChannel是通过调用InputQueue类的静态成员函数unregisterInputChannel来注销的，接下来我们就从这个函数开始分析一个Client端InputChannel的注销过程，如图14-21所示。

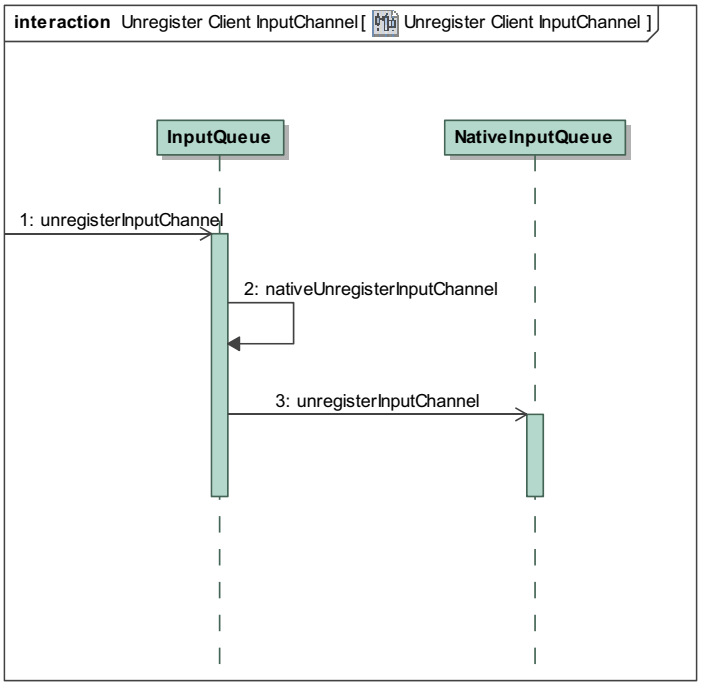


图14-21 Client端InputChannel的注销过程

这个过程一共分为3个步骤，下面我们就详细分析每一个步骤。

Step 1：InputQueue.unregisterInputChannel

frameworks/base/core/java/android/view/InputQueue.java

```
01 public final class InputQueue {
02     ......
03
04     private static native void nativeUnregisterInputChannel(InputChannel inputChannel);
```

```
05      ......
06
07      public static void unregisterInputChannel(InputChannel inputChannel) {
08          ......
09
10          synchronized (sLock) {
11              ......
12
13              nativeUnregisterInputChannel(inputChannel);
14          }
15      }
16
17      ......
18  }
```

第13行代码调用了InputQueue类的静态成员函数nativeUnregisterInputChannel来注销参数inputChannel所描述的一个Client端InputChannel。

Step 2：InputQueue.nativeUnregisterInputChannel

frameworks/base/core/jni/android_view_InputQueue.cpp

```
1 static void android_view_InputQueue_nativeUnregisterInputChannel(JNIEnv* env, jclass clazz,
2          jobject inputChannelObj) {
3      status_t status = gNativeInputQueue.unregisterInputChannel(env, inputChannelObj);
4
5      ......
6 }
```

InputQueue类的静态成员函数nativeUnregisterInputChannel是一个JNI方法，它是由C++层的函数android_view_InputQueue_nativeUnregisterInputChannel来实现的。

从前面14.3.4小节的Step 2可以知道，gNativeInputQueue是一个全局变量，它指向了一个NativeInputQueue对象，第3行代码调用它的成员函数unregisterInputChannel来注销参数inputChannelObj所描述的一个Client端InputChannel。

Step 3：NativeInputQueue.unregisterInputChannel

frameworks/base/core/jni/android_view_InputQueue.cpp

```
01 status_t NativeInputQueue::unregisterInputChannel(JNIEnv* env, jobject inputChannelObj) {
02      sp<InputChannel> inputChannel = android_view_InputChannel_getInputChannel(env,
03          inputChannelObj);
04      ......
05
06      { // acquire lock
07          AutoMutex _l(mLock);
08
09          ssize_t connectionIndex = getConnectionIndex(inputChannel);
10          ......
11
12          sp<Connection> connection = mConnectionsByReceiveFd.valueAt(connectionIndex);
13          mConnectionsByReceiveFd.removeItemsAt(connectionIndex);
14          ......
15
16          connection->looper->removeFd(inputChannel->getReceivePipeFd());
17
18          env->DeleteGlobalRef(connection->inputHandlerObjGlobal);
19          connection->inputHandlerObjGlobal = NULL;
20
21          ......
22      } // release lock
23
24      ......
25      return OK;
26 }
```

参数inputChannelObj指向了一个Java层的Client端InputChannel，第2行和第3行代码首先调用函数android_view_InputChannel_getInputChannel将它转换为一个C++层的Client端InputChannel，并且保存在变量inputChannel中。

从前面14.3.4小节的Step 3可以知道，一个C++层的Client端InputChannel在注册时，首先会被封装成一个Connection对象，然后再以这个Client端InputChannel内部的一个前向管道的读端文件描述符为关键字，将前面所封装的一个Connection对象保存在NativeInputQueue类的成员变量mConnectionsByReceiveFd中。因此，在注销这个Client端InputChannel时，就需要将与它所对应的Connection对象从NativeInputQueue类的成员变量mConnectionsByReceiveFd中移除。

第9行到第13行代码首先找到变量inputChannel所描述的一个Client端InputChannel在NativeInputQueue类的成员变量mConnectionsByReceiveFd中的位置connectionIndex，以及它所对应的Connection对象connection，然后再将找到的Connection对象从NativeInputQueue类的成员变量mConnectionsByReceiveFd中移除。

从前面14.3.4小节的Step 3还可以知道，一个C++层的Client端InputChannel在注册时，它内部的一个前向管道的读端文件描述符会被注册到一个Looper对象中，以便与这个Looper所关联的一个应用程序主线程可以监控这个前向管道的IO写事件。因此，在注销这个Client端InputChannel时，就需要将它内部的一个前向管道的读端文件描述符从对应的Looper对象中移除。

变量inputChannel所描述的一个Client端InputChannel内部的一个前向管道的读端文件描述符所注册在的Looper对象，就保存在与它所对应的Connection对象connection的成员变量looper中。因此，第16行代码首先获得这个Client端InputChannel内部的一个前向管道的读端文件描述符，接着再将这个文件描述符从Connection对象connection的成员变量looper所描述的一个Looper对象中移除。

从前面14.3.4小节的Step 3又可以知道，一个C++层的Client端InputChannel在注册时，与它所关联的一个Connection对象的成员变量inputHandlerObjGlobal引用了一个Java层的InputHandler对象。因此，在注销这个Client端InputChannel时，就需要删除这个引用，以便被引用的Java层的InputHandler对象可以得到释放。

最后，第18行和第19行代码首先删除Connection对象connection的成员变量inputHandlerObjGlobal所描述的一个Java层的InputHandler对象引用，接着再将它的值设置为null。这一步执行完成之后，与变量inputChannel所描述的一个Client端InputChannel所对应的一个Connection对象的清理工作就完成了。

至此，一个Client端InputChannel的注销过程就分析完成了。接下来我们继续分析一个Server端InputChannel的注销过程。

14.5.3 注销Server端InputChannel

一个应用程序窗口在创建时，会被添加到Window管理服务WindowManagerService中，因此，当这个应用程序窗口被销毁时，就需要从Window管理服务WindowManagerService中移除。Window管理服务WindowManagerService在移除一个应用程序窗口的过程中，会将与它所关联的一个Server端InputChannel从系统输入管理器InputManager中注销。

从前面14.5.1小节的Step 11可以知道，将一个应用程序窗口从Window管理服务WindowManagerService中移除是通过调用Session类的成员函数remove来实现的，接下来我们就从这个函数开始分析一个Server端InputChannel的注销过程，如图14-22所示。

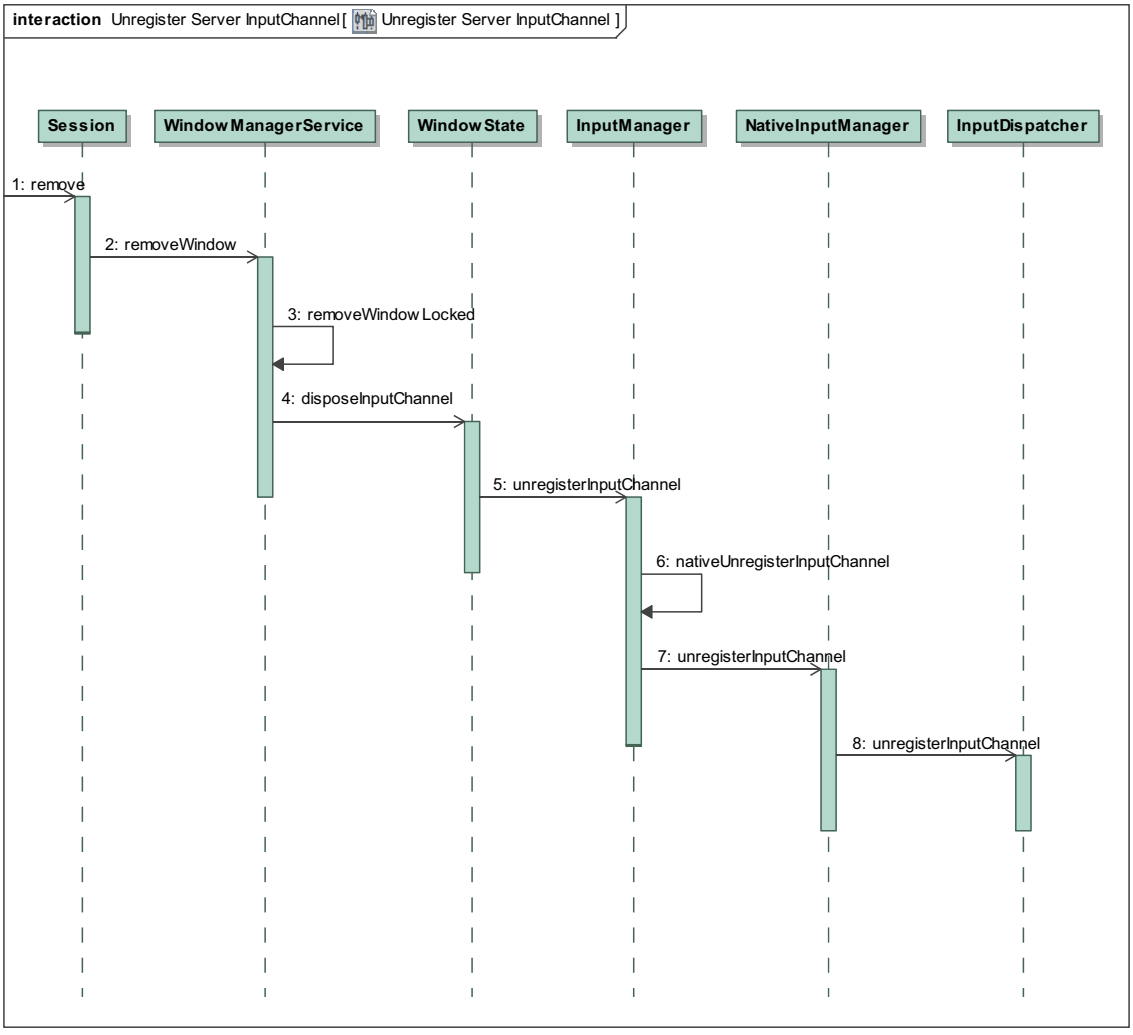


图14-22　Server端InputChannel的注销过程

这个过程一共分为8个步骤，下面我们就详细分析每一个步骤。

Step 1：Session.remove

frameworks/base/services/java/com/android/server/WindowManagerService.java

```
01 public class WindowManagerService extends IWindowManager.Stub
02         implements Watchdog.Monitor {
03     ......
04
05     private final class Session extends IWindowSession.Stub
06             implements IBinder.DeathRecipient {
07         ......
08
09         public void remove(IWindow window) {
10             removeWindow(this, window);
11         }
12
13         ......
14     }
15
16     ......
17 }
```

参数window用来描述一个正在销毁的应用程序窗口，第10行代码调用了外部类WindowManagerService的成员函数removeWindow将它从Window管理服务WindowManagerService中移除。

Step 2：WindowManagerService.removeWindow

frameworks/base/services/java/com/android/server/WindowManagerService.java

```
01 public class WindowManagerService extends IWindowManager.Stub
02         implements Watchdog.Monitor {
03     ......
04
05     public void removeWindow(Session session, IWindow client) {
06         synchronized(mWindowMap) {
07             WindowState win = windowForClientLocked(session, client, false);
08             if (win == null) {
09                 return;
10             }
11             removeWindowLocked(session, win);
12         }
13     }
14
15     ......
16 }
```

从前面14.3.1小节的Step 3可以知道，一个应用程序窗口在创建时，会被封装成一个WindowState对象，并且保存在Window管理服务WindowManagerService的成员变量mWindowMap和mWindows中。因此，当这个应用程序窗口被销毁时，就需要将与它所对应的WindowState对象从Window管理服务WindowManagerService的成员变量mWindowMap和mWindows中移除。

参数client描述的是一个正在销毁的应用程序窗口，第7行代码首先找到与它所对应的一个WindowState对象，接着第11行代码调用WindowManagerService类的成员函数removeWindowLocked来移除这个WindowState对象，这样就相当于将正在销毁的应用程序窗口从Window管理服务WindowManagerService中移除了。

Step 3：WindowManagerService.removeWindowLocked

frameworks/base/services/java/com/android/server/WindowManagerService.java

```
01 public class WindowManagerService extends IWindowManager.Stub
02         implements Watchdog.Monitor {
03     ......
04
05     public void removeWindowLocked(Session session, WindowState win) {
06         ......
07
08         win.disposeInputChannel();
09         ......
10
11         removeWindowInnerLocked(session, win);
12         ......
13     }
14
15     ......
16 }
```

第8行代码首先调用参数win所描述的一个WindowState对象的成员函数disposeInputChannel来注销与它所关联的一个Server端InputChannel，接着第11行代码调用WindowManagerService类的成员函数removeWindowInnerLocked将这个WindowState对象从Window管理服务WindowManagerService的成员变量mWindowMap和mWindows中移除。

接下来，我们继续分析WindowState类的成员函数disposeInputChannel的实现，以便可以了解它是如何注销一个Server端InputChannel的。

Step 4：WindowState.disposeInputChannel

frameworks/base/services/java/com/android/server/WindowManagerService.java

```
01 public class WindowManagerService extends IWindowManager.Stub
02         implements Watchdog.Monitor {
03     ......
04
05     final InputManager mInputManager;
06     ......
07
08     private final class WindowState implements WindowManagerPolicy.WindowState {
09         ......
10
11         InputChannel mInputChannel;
12         ......
13
14         void disposeInputChannel() {
15             if (mInputChannel != null) {
16                 mInputManager.unregisterInputChannel(mInputChannel);
17
18                 mInputChannel.dispose();
19                 mInputChannel = null;
20             }
21         }
22
23         ......
24     }
25
26     ......
27 }
```

从前面14.3.1小节的Step 3可以知道，WindowState类的成员变量mInputChannel指向了一个Server端InputChannel，第16行代码调用系统输入管理器InputManager的成员函数unregisterInputChannel将这个Server端InputChannel注销，因为前面我们将它注册到系统输入管理器InputManager中了。

Step 5：InputManager.unregisterInputChannel

frameworks/base/services/java/com/android/server/InputManager.java

```
01 public class InputManager {
02     ......
03
04     private static native void nativeUnregisterInputChannel(InputChannel inputChannel);
05     ......
06
07     public void unregisterInputChannel(InputChannel inputChannel) {
08         ......
09
10         nativeUnregisterInputChannel(inputChannel);
11     }
12
13     ......
14 }
```

第10行代码调用InputManager类的静态成员函数nativeUnregisterInputChannel来注销参数inputChannel所描述的一个Server端InputChannel。

Step 6：InputManager.nativeUnregisterInputChannel

frameworks/base/services/jni/com_android_server_InputManager.cpp

```
01 static void android_server_InputManager_nativeUnregisterInputChannel(JNIEnv* env, jclass clazz,
02         jobject inputChannelObj) {
03     ......
04
```

```
05     sp<InputChannel> inputChannel = android_view_InputChannel_getInputChannel(env,
06                         inputChannelObj);
07     ......
08
09     status_t status = gNativeInputManager->unregisterInputChannel(env, inputChannel);
10
11     ......
12 }
```

InputManager类的静态成员函数nativeUnregisterInputChannel是一个JNI方法，它是由C++层的函数android_server_InputManager_nativeUnregisterInputChannel来实现的。

参数inputChannelObj指向的是一个Java层的Server端InputChannel，第5行和第6行代码首先调用函数android_view_InputChannel_getInputChannel将它转换为一个C++层的Server端InputChannel，接着第9行代码再调用全局变量gNativeInputManager的成员函数unregisterInputChannel来注销这个C++层的Server端InputChannel。

从前面14.2.1小节的Step 3可以知道，全局变量gNativeInputManager指向了一个NativeInputManager对象。因此，接下来我们就继续分析NativeInputManager类的成员函数unregisterInputChannel的实现。

Step 7：NativeInputManager.unregisterInputChannel

frameworks/base/services/jni/com_android_server_InputManager.cpp

```
1 status_t NativeInputManager::unregisterInputChannel(JNIEnv* env,
2         const sp<InputChannel>& inputChannel) {
3     ......
4
5     return mInputManager->getDispatcher()->unregisterInputChannel(inputChannel);
6 }
```

从前面14.2.1小节的Step 3可以知道，NativeInputManager类的成员变量mInputManager指向了一个C++层的InputManager对象，第5行代码首先调用它的成员函数getDispatcher来获得它内部的一个InputDispatcher对象，接着再调用这个InputDispatcher对象的成员函数unregisterInputChannel来注销参数inputChannel所描述的一个Server端InputChannel。

Step 8：InputDispatcher.unregisterInputChannel

frameworks/base/libs/ui/InputDispatcher.cpp

```
01 status_t InputDispatcher::unregisterInputChannel(const sp<InputChannel>& inputChannel) {
02     ......
03
04     { // acquire lock
05         AutoMutex _l(mLock);
06
07         ssize_t connectionIndex = getConnectionIndexLocked(inputChannel);
08         ......
09
10         mConnectionsByReceiveFd.removeItemsAt(connectionIndex);
11         ......
12
13         mLooper->removeFd(inputChannel->getReceivePipeFd());
14
15         .....
16
17     } // release lock
18
19     ......
20     return OK;
21 }
```

从前面14.3.2小节的Step 4可以知道，一个Server端InputChannel在注册时，首先会被封装成一个Connection对象，然后再以这个Server端InputChannel内部的一个反向管道的读端文件描述符为关键字，将前面所封装的一个Connection对象保存在InputDispatcher类的成员变量mConnectionsByReceiveFd中。因此，在注销这个Server端InputChannel时，就需要将与它所对应的Connection对象从InputDispatcher类的成员变量mConnectionsByReceiveFd中移除。

第7行和第10行代码首先找到参数inputChannel所描述的一个Server端InputChannel在InputDispatcher类的成员变量mConnectionsByReceiveFd中的位置connectionIndex，然后再根据这个位置将与它所对应的一个Connection对象从InputDispatcher类的成员变量mConnectionsByReceiveFd中移除。

从前面14.3.2小节的Step 4还可以知道，一个Server端InputChannel在注册时，它内部的一个反向管道的读端文件描述符会被注册到与InputDispatcher所运行在的线程所关联的一个Looper对象中，以便InputDispatcher可以监控这个反向管道的IO写事件。因此，在注销这个Server端InputChannel时，就需要将它内部的一个反向管道的读端文件描述符从与InputDispatcher所运行在的线程所关联的一个Looper对象中移除。

InputDispatcher类的成员变量mLooper描述的便是与InputDispatcher所运行在的线程所关联的一个Looper对象。因此，第13行代码首先获得参数inputChannel所描述的一个Server端InputChannel的内部的一个反向管道的读端文件描述符，然后再将这个文件描述符从InputDispatcher类的成员变量mLooper所描述的一个Looper对象中移除。这一步执行完成之后，参数inputChannel所描述的一个Server端InputChannel的注销工作就完成了。

至此，一个Server端InputChannel的注销过程就分析完成了，Android应用程序的键盘消息处理机制也分析完成了。理解了Android应用程序的键盘消息处理机制之后，我们就可以对Android应用程序的整体架构有一个更加清晰的认识了。

第 15 章

Android应用程序线程的消息循环模型

我们知道，Android应用程序是支持多线程的。Android应用程序线程有主线程和子线程之分，其中，主线程是由Activity管理服务ActivityManagerService请求Zygote进程创建的；而子线程是由主线程或者其他子线程创建的。

从前面第12章的内容可以知道，Android应用程序主线程在启动完成之后，会主动进入到一个消息循环中，即Android应用程序主线程是具有消息循环的。那么Android应用程序子线程能不能像主线程一样也具有消息循环呢？在回答这个问题之前，我们首先分析Android应用程序子线程为什么需要消息循环。

Android应用程序的主线程又称为界面线程[1]，它是负责响应界面事件的。因此，我们要避免在它里面执行耗时的操作，因为执行耗时的操作会导致主线程长时间不能响应界面事件，从而影响用户体验。我们一般把一个耗时的操作看作是一个后台任务，并且放在一个子线程中执行。在这些后台任务中，有的是一次性的，即执行了一次之后就不再需要了；而有的是需要不定期执行的，即条件满足时就执行，条件不满足时就不执行。

对于不定期的后台任务来说，它们有两种执行方式。第一种方式是每当条件满足时，就创建一个子线程来执行一个不定期的后台任务；而当这个不定期的任务执行完成之后，新创建的子线程就随之退出。第二种方式是创建一个具有消息循环的子线程，每当条件满足时，就将一个不定期的后台任务封装成一个消息发送到这个子线程的消息队列中去执行；而当条件不满足时，这个子线程就会因为它的消息队列为空而进入睡眠等待状态。虽然第一种方式创建的子线程不需要具有消息循环，但是不断地创建和销毁子线程是有代价的，因此，我们更倾向于采用第二种方式来执行那些不定期的后台任务。从这个角度来看，我们就希望Android应用程序子线程像主线程一样具有消息循环。

无论是对于一次性的后台任务来说，还是对于定期性的后台任务来说，它们在执行的过程中可能都会涉及界面操作。例如，我们通常把一个下载任务放在一个子线程中来执行，而在下载的过程中，一般需要将下载进度显示在应用程序的界面上。在这种情况下，用来执行它们的子线程一般就会将一个与界面相关的操作封装成一个消息，并且发送到主线程的消息队列中，以便主线程在处理这个消息时，可以执行一个相关的界面操作。为了简化子线程执行与界面相关的操作，我们通常将它们与主线程的消息循环关联起来。从这个角度来看，我们也可以说Android应用程序子线程是需要有消息循环的。

综上所述，我们就可以得到Android应用程序线程的三种消息循环模型。第一种是应用程序主线程消息循环模型；第二种是与界面无关的应用程序子线程消息循环模型；第三种是与界面相关的应用程序子线程消息循环模型。我们可以使用ActivityThread、HandlerThread和AsyncTask这三个类来分别实现上述三种消息循环模型。

1 在Android系统中，界面操作是非线程安全的，因此，Android系统规定只可以在Android应用程序的主线程中执行与界面相关的操作。

接下来，我们首先分析应用程序主线程消息循环模型，然后分析与界面无关的应用程序子线程消息循环模型，最后分析与界面相关的应用程序子线程消息循环模型。

15.1　应用程序主线程消息循环模型

从前面第12章的内容可以知道，Activity管理服务ActivityManagerService在启动一个应用程序组件时，如果发现这个应用程序组件需要在一个新的应用程序进程中运行，那么就会调用Process类的静态成员函数start来请求Zygote进程创建一个新的应用程序进程。

Process类的静态成员函数start的实现如下所示。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02         implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03     ......
04
05     private final void startProcessLocked(ProcessRecord app,
06                 String hostingType, String hostingNameStr) {
07         ......
08
09         try {
10             ......
11
12             int pid = Process.start("android.app.ActivityThread",
13                 mSimpleProcessManagement ? app.processName : null, uid, uid,
14                 gids, debugFlags, null);
15
16             ......
17         } catch (RuntimeException e) {
18             ......
19         }
20     }
21
22     ......
23 }
```

Process类的静态成员函数start的第一个参数用来指定新创建的应用程序进程在创建完成之后需要加载的类，并且是以这个类的静态成员函数main来作为新创建的应用程序进程的入口函数的。从第12行代码可以看出，每一个新创建的应用程序进程都是以ActivityThread类的静态成员函数main来作为入口函数的。由于一个新创建的应用程序进程即为一个Android应用程序的主线程，因此，我们也可以说Android应用程序的主线程是以ActivityThread类的静态成员函数main来作为入口函数的。

ActivityThread类的静态成员函数main的实现如下所示。

frameworks/base/core/java/android/app/ActivityThread.java

```
01 public final class ActivityThread {
02     ......
03
04     public static final void main(String[] args) {
05         ......
06
07         Looper.prepareMainLooper();
08         ......
09
10         Looper.loop();
11
12         ......
13     }
14
15     ......
16 }
```

第7行代码首先调用Looper类的静态成员函数prepareMainLooper在当前应用程序主线程中创建一个消息循环，接着第10行代码再调用Looper类的成员函数loop使得当前应用程序主线程进入到前面所创建的一个消息循环中。

从前面13.1小节的内容可以知道，Looper类的静态成员函数prepareMainLooper首先会创建一个Looper对象，用来描述一个应用程序主线程的消息循环；接着又会在这个Looper对象中创建一个MessageQueue对象，用来描述一个应用程序主线程的消息队列。

注意

Looper类的静态成员函数prepareMainLooper只能在应用程序的主线程中调用，并且它只能被调用一次。由于Looper类的静态成员函数prepareMainLooper在应用程序主线程启动时已经被调用过了，因此，以后无论是在应用程序的主线程中，还是在应用程序的子线程中，我们都不应该再次调用它；否则，系统就会抛出一个运行时异常。在接下来的15.2小节中，我们就会看到，一个应用程序子线程是通过调用Looper类的静态成员函数prepare来创建一个消息循环的。

为什么Looper类要单独为应用程序主线程提供一个创建Looper对象的成员函数呢？从前面13.1小节的内容可以知道，这个Looper对象除了会保存在一个线程局部变量中之外，还会单独保存在Looper类的静态成员变量mMainLooper中。这样我们就可以在应用程序子线程中调用Looper类的静态成员函数getMainLooper来获得这个Looper对象，并且通过这个Looper对象向应用程序主线程的消息队列发送与界面操作相关的消息，用来解决应用程序子线程不能直接操作应用程序界面的问题。

至此，应用程序主线程的消息循环模型就分析完成了。接下来我们继续分析与界面无关的应用程序子线程的消息循环模型。

15.2 与界面无关的应用程序子线程消息循环模型

Android应用程序子线程是使用Thread类来描述的，不过我们一般不直接使用Thread类来创建一个Android应用程序子线程，而是首先实现一个Thread子类，然后使用这个Thread子类来创建一个Android应用程序子线程，以便可以在重写的成员函数run中实现子线程的程序逻辑。

接下来，我们通过一个具体的例子来说明如何使用一个Thread子类来创建一个Android应用程序子线程。

首先定义一个Thread子类SubThread，如下所示。

```
01 public class SubThread extends Thread {
02     // constrcutor
03     public SubThread(String name) {
04         super(name);
05     }
06
07     // execute in the sub thread
08     public void run() {
09         // start
10         // execute the task
11         // exit
12     }
13 }
```

接着再创建一个SubThread类实例，并且调用它的成员函数start来启动一个子线程，如下所示。

```
1 SubThread subThread = new SubThread("Sub Thread");
2 subThread.start();
```

当线程subThread被创建完成之后，SubThread类的成员函数run就会被调用，这样我们就可以在里面实现自己的程序逻辑了。

注意

使用上述步骤创建的应用程序子线程是没有消息循环的，当它的成员函数run执行完成之后，该线程就会随之结束。

为了创建一个具有消息循环的应用程序子线程，我们可以通过Android应用程序框架层提供的HandlerThread类来实现，它的实现如下所示。

frameworks/base/core/java/android/os/HandlerThread.java

```
01 public class HandlerThread extends Thread {
02     ......
03
04     private Looper mLooper;
05     ......
06
07     public HandlerThread(String name) {
08         super(name);
09         ......
10     }
11     ......
12
13     public void run() {
14         ......
15
16         Looper.prepare();
17         synchronized (this) {
18             mLooper = Looper.myLooper();
19             ......
20         }
21         ......
22
23         Looper.loop();
24
25         ......
26     }
27     ......
28
29     public Looper getLooper() {
30         ......
31         return mLooper;
32     }
33     ......
34
35     public boolean quit() {
36         Looper looper = getLooper();
37         if (looper != null) {
38             looper.quit();
39             return true;
40         }
41         return false;
42     }
43
44     ......
45 }
```

与SubThread类一样，HandlerThread类也是一个Thread子类，不过它的成员函数run首先会调用Looper类的静态成员函数prepare来创建一个消息循环，接着再调用Looper类的静态成员函数loop使得当前线程进入到前面所创建的一个消息循环中。

下面我们就通过一个具体的例子来说明如何使用HandlerThread类创建一个Android应用程序子线程。

首先创建一个HandlerThread类实例，并且调用它的成员函数start来启动一个子线程，如下所示。

```
1 HandlerThread handlerThread = new HandlerThread("Handler Thread");
2 handlerThread.start();
```

然后定义一个实现了Runnable接口的类，例如ThreadTask类，用来描述一个需要在前面所创建的子线程中执行的任务，如下所示。

```
01 public class ThreadTask implements Runnable {
02     // constructor
03     public ThreadTask() {
04
05     }
06
07     // execute in the handler thread
08     public void run() {
09         // start
10         // execute the task
11         // exit
12     }
13 }
```

接着创建一个ThreadTask类实例，如下所示。

```
1 ThreadTask threadTask = new ThreadTask();
```

最后就可以将这个ThreadTask类实例封装成一个消息发送到前面所创建的一个子线程的消息队列中了，如下所示。

```
1 Handler handler = new Handler(handlerThread.getLooper());
2 handler.post(threadTask);
```

当这个消息被处理时，前面所创建的一个子线程就会调用ThreadTask类的成员函数run，以便它可以执行一个自定义的任务。

注意

如果我们不往前面所创建的一个子线程的消息队列发送消息，那么它就会因为它的消息队列为空而进入睡眠等待状态。

当我们需要结束执行前面所创建的一个子线程时，只要调用它的成员函数quit就可以了，如下所示。

```
1 handlerThread.quit();
```

如前所示，HandlerThread类的成员函数quit首先获得前面在子线程中所创建的一个Looper对象，然后再调用这个Looper对象的成员函数quit来退出子线程。Looper类的成员函数quit的实现如下所示。

frameworks/base/core/java/android/os/Looper.java

```
01 public class Looper {
02     ......
03
04     final MessageQueue mQueue;
05     ......
06
07     public void quit() {
08         Message msg = Message.obtain();
09         // NOTE: By enqueueing directly into the message queue, the
10         // message is left with a null target.  This is how we know it is
11         // a quit message.
12         mQueue.enqueueMessage(msg, 0);
13     }
14
15     ......
16 }
```

从前面13.4小节的Step 1可以知道，如果我们向一个线程的消息队列发送一个成员变量target为null的Message对象，那么这个线程就会退出消息循环。因此，我们就可以停止执行这个线程。

至此，与界面无关的应用程序子线程的消息循环模型就分析完成了。接下来我们继续分析与界面相关的应用程序子线程的消息循环模型。

15.3　与界面相关的应用程序子线程消息循环模型

Android系统在应用程序框架层中提供了一个异步任务类AsyncTask，用来将一个涉及界面操作的任务放在一个子线程中执行。用来执行异步任务的子线程虽然没有自己的消息循环，但是它会利用主线程的消息循环来执行与界面相关的操作。接下来，我们就结合一个具体的例子来分析异步任务类AsyncTask的实现，以便可以了解与界面相关的应用程序子线程的消息循环模型。

在前面的8.1小节中，我们开发了一个CounterService组件，当它的成员函数startCounter被调用时，它就会在一个子线程中执行一个异步任务。这个异步任务每隔1秒就会将它内部的一个计数值增加1，并且将这个计数值更新到应用程序的界面上，这样就可以实现一个计数器服务了。

CounterService组件的成员函数startCounter的实现如下所示。

packages/experimental/Counter/src/shy/luo/counter/CounterService.java

```
01 public class CounterService extends Service implements ICounterService {
02     ......
03
04     private ICounterCallback counterCallback = null;
05     ......
06
07     public void startCounter(int initVal, ICounterCallback callback) {
08         counterCallback = callback;
09
10         AsyncTask<Integer, Integer, Integer> task = new AsyncTask<Integer, Integer, Integer>() {
11             @Override
12             protected Integer doInBackground(Integer... vals) {
13                 Integer initCounter = vals[0];
14
15                 stop = false;
16                 while(!stop) {
17                     publishProgress(initCounter);
18
19                     try {
20                         Thread.sleep(1000);
21                     } catch (InterruptedException e) {
22                         e.printStackTrace();
23                     }
24
25                    initCounter++;
26                }
27
28                return initCounter;
29            }
30
31            @Override
32            protected void onProgressUpdate(Integer... values) {
33                super.onProgressUpdate(values);
34
35                int val = values[0];
36                counterCallback.count(val);
37            }
38
39            @Override
```

```
40              protected void onPostExecute(Integer val) {
41                  counterCallback.count(val);
42              }
43
44          };
45
46          task.execute(initVal);
47      }
48
49      ......
50  }
```

参数initVal是一个计数器的初始值；而参数callback是一个计数器回调接口，通过调用它的成员函数count就可以将一个计数值更新到应用程序的界面上。

第8行代码首先将参数callback所描述的一个计数器回调接口保存在CounterService组件的成员变量counterCallback中；然后第10行到第44行代码创建了一个类型为AsyncTask<Integer, Integer, Integer>的异步任务task；最后第46行代码将这个异步任务启动起来，以便它可以执行定时计数的功能。

异步任务task是在AsyncTask<Integer, Integer, Integer>类的成员函数doInBackground中执行的，并且它是运行在一个子线程中的。因此，它在执行的过程中不会影响主线程处理界面事件。

AsyncTask<Integer, Integer, Integer>类的成员函数doInBackground每隔1秒就会将内部的一个计数值增加1，并且会调用另外一个成员函数publishProgress将这个计数值分发给第三个成员函数onProgressUpdate来处理，后者通过保存在CounterService组件的成员变量counterCallback中的一个计数器回调接口将接收到的计数值更新到应用程序的界面上。

异步任务task执行完成之后，就会从AsyncTask<Integer, Integer, Integer>类的成员函数doInBackground返回来，并且将它的返回值分发给成员函数onPostExecute来处理。这个返回值即为异步任务task所描述的一个计数器的终止值，AsyncTask<Integer, Integer, Integer>类的成员函数onPostExecute同样是通过保存在CounterService组件的成员变量counterCallback中的一个计数器回调接口将它更新到应用程序的界面上的。

由于AsyncTask<Integer, Integer, Integer>类的成员函数onProgressUpdate和onPostExecute执行了与界面相关的操作，因此，我们就可以推断出它们是在应用程序的主线程中执行的；否则，系统就会抛出一个界面异常。从这里就可以看出，一个异步任务虽然是在一个子线程中执行的，但是它可以执行与界面相关的操作，接下来我们就分析它是如何做到这一点的。

我们首先了解异步任务类AsyncTask的成员变量的定义，如下所示。

frameworks/base/core/java/android/os/AsyncTask.java

```
01 public abstract class AsyncTask<Params, Progress, Result> {
02     ......
03
04     private static final BlockingQueue<Runnable> sWorkQueue =
05             new LinkedBlockingQueue<Runnable>(10);
06
07     private static final ThreadFactory sThreadFactory = new ThreadFactory() {
08         private final AtomicInteger mCount = new AtomicInteger(1);
09
10         public Thread newThread(Runnable r) {
11             return new Thread(r, "AsyncTask #" + mCount.getAndIncrement());
12         }
13     };
14     ......
15
16     private static final ThreadPoolExecutor sExecutor = new ThreadPoolExecutor(CORE_POOL_SIZE,
17         MAXIMUM_POOL_SIZE, KEEP_ALIVE, TimeUnit.SECONDS, sWorkQueue, sThreadFactory);
18
19     private static final int MESSAGE_POST_RESULT = 0x1;
20     private static final int MESSAGE_POST_PROGRESS = 0x2;
```

```
21     private static final int MESSAGE_POST_CANCEL = 0x3;
22
23     private static final InternalHandler sHandler = new InternalHandler();
24
25     private final WorkerRunnable<Params, Result> mWorker;
26     private final FutureTask<Result> mFuture;
27
28     ......
29 }
```

异步任务类AsyncTask是一个泛类，它有三个类型参数Params、Progress和Result，分别用来描述一个异步任务的输入数据、过程数据和结果数据的类型。

异步任务类AsyncTask的静态成员变量sWorkQueue指向了一个类型为LinkedBlockingQueue的工作任务队列，保存在这个队列中的每一个任务都是使用一个实现了Runnable接口的对象来描述的。

一个类型为LinkedBlockingQueue的工作任务队列有以下两个特性。

（1）一个线程如果试图从一个空的LinkedBlockingQueue队列中取出一个工作任务来执行，那么它就会被阻塞，直到这个队列有新的工作任务添加进来为止。

（2）一个线程如果试图往一个满的LinkedBlockingQueue队列中添加一个新的工作任务，那么它同样会被阻塞，直到这个队列腾出来新的空闲位置为止。

异步任务类AsyncTask的静态成员变量sThreadFactory指向了一个类型为ThreadFactory的线程创建工厂，它创建出来的线程是用来执行保存在静态成员变量sWorkQueue中的工作任务的。

异步任务类AsyncTask的静态成员变量sExecutor指向了一个类型为ThreadPoolExecutor的线程池，保存在这个线程池中的线程就是通过静态成员变量sThreadFactory来创建的，并且用来执行保存在静态成员变量sWorkQueue中的工作任务。接下来，我们就继续分析ThreadPoolExecutor线程池的工作原理，以便可以更好地理解异步任务的实现原理。

ThreadPoolExecutor类的构造函数的原型如下所示。

```
1 ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
2     long keepAliveTime, TimeUnit unit,
3     BlockingQueue<Runnable> workQueue,
4     ThreadFactory threadFactory);
```

参数corePoolSize和maximumPoolSize用来描述一个ThreadPoolExecutor线程池的核心线程数量和最大线程数量。

当一个ThreadPoolExecutor线程池的线程数量大于核心线程数量时，保存在它里面的空闲线程就会被设置一个存活时间，其中，时间长度由参数keepAliveTime来描述，而时间单位由参数unit来描述。

最后，参数workQueue和threadFactory分别用来描述一个ThreadPoolExecutor线程池的工作任务队列和线程创建工厂。

当我们需要在一个 ThreadPoolExecutor线程池中执行一个工作任务时，首先需要将要执行的工作任务封装成一个实现了Runnable接口的对象，接着再调用 ThreadPoolExecutor类的成员函数execute将这个对象添加到这个 ThreadPoolExecutor线程池中去执行。

ThreadPoolExecutor类的成员函数execute的工作方式如下所示。

（1）如果线程池中的线程数量小于核心线程数量，那么它就会创建一个新的线程来执行新添加进来的工作任务。

（2）如果线程池中的线程数量等于或者大于核心线程数量，但是小于最大线程数量，这时候：

- 如果工作任务队列处于未满的状态，那么它就会将新添加进来的工作任务保存在工作任务队列中等待执行。
- 如果工作任务队列处于已满的状态，那么它就会创建一个新的线程来执行新添加进来的工作任务。

（3）如果线程池中的线程量等于或者大于最大线程数量，这时候：

- 如果工作任务队列处于未满的状态，那么它就会将新添加进来的工作任务保存在工作任务队列中等待执行。
- 如果工作任务队列处于已满的状态，那么它就会拒绝执行新添加进来的工作任务。

由于异步任务类AsyncTask的成员变量sWorkQueue、sThreadFactory和sExecutor都是静态成员变量，因此，在同一个应用程序进程中，所有使用异步任务类AsyncTask来描述的异步任务都是在同一个线程池中执行的。这样就可以避免在同一个应用程序进程中重复地创建线程池，毕竟创建线程池是需要代价的。

异步任务类AsyncTask的静态成员变量MESSAGE_POST_RESULT、MESSAGE_POST_PROGRESS和MESSAGE_POST_CANCEL是三个常量，它们分别用来描述三个不同的消息类型，后面我们再分析它们的作用。

异步任务类AsyncTask的静态成员变量sHandler指向了一个类型为InternalHandler的Handler的对象。

注意

由于这个Handler对象是异步任务类AsyncTask的一个静态成员变量，因此，它是在异步任务类AsyncTask第一次被应用程序使用时创建的。假设应用程序第一次使用异步任务类AsyncTask时是运行在主线程中的，这时候异步任务类AsyncTask的静态成员变量sHandler所指向的一个Handler对象就是在应用程序的主线程中创建的，这意味着它可以用来向应用程序主线程的消息队列发送消息。

最后，异步任务类AsyncTask的成员变量mWorker和mFuture用来描述一个工作任务，它们的类型分别为WorkerRunnable<Params, Result>和FutureTask<Result>，后面我们再分析它们的作用。

通过前面的描述可以知道，InternalHandler类是与应用程序主线程的消息队列相关的，而WorkerRunnable类是与工作任务相关的。接下来我们就继续分析它们的实现，以便可以进一步了解异步任务类AsyncTask的静态成员变量sHandler，以及非静态成员变量mWorker的含义。

InternalHandler类继承了Handler类，并且重写了Handler类的成员函数handleMessage，它的实现如下所示。

frameworks/base/core/java/android/os/AsyncTask.java

```
01 public abstract class AsyncTask<Params, Progress, Result> {
02     ......
03
04     private static class InternalHandler extends Handler {
05         @SuppressWarnings({"unchecked", "RawUseOfParameterizedType"})
06         @Override
07         public void handleMessage(Message msg) {
08             AsyncTaskResult result = (AsyncTaskResult) msg.obj;
09             switch (msg.what) {
10                 case MESSAGE_POST_RESULT:
11                  // There is only one result
12                  result.mTask.finish(result.mData[0]);
13                  break;
14                 case MESSAGE_POST_PROGRESS:
15                  result.mTask.onProgressUpdate(result.mData);
16                  break;
17                 case MESSAGE_POST_CANCEL:
18                  result.mTask.onCancelled();
19                  break;
20             }
21         }
22     }
23
24     ......
25 }
```

InternalHandler类的成员函数handleMessage主要用来处理MESSAGE_POST_RESULT、MESSAGE_POST_PROGRESS和MESSAGE_POST_CANCEL三种类型的消息。这三种类型的消息都是由执行异步任务的子线程发送到应用程序主线程的消息队列中的，它们的发送情景如下所示。

（1）一个异步任务在执行的过程中，会不断地向应用程序主线程的消息队列发送类型为MESSAGE_POST_PROGRESS的消息，主要用来向应用程序的主线程报告它的执行情况。

（2）一个异步任务在执行完成时，就会向应用程序主线程的消息队列发送一个类型为MESSAGE_POST_RESULT的消息，以便可以通知应用程序的主线程它执行完成了。

（3）一个异步任务在被取消执行时，就会向应用程序主线程的消息队列发送一个类型为MESSAGE_POST_CANCEL的消息，以便可以通知应用程序的主线程它被取消执行了。

在InternalHandler类的成员函数handleMessage中，参数msg的成员变量obj指向了一个AsyncTaskResult对象，它的实现如下所示。

frameworks/base/core/java/android/os/AsyncTask.java

```
01 public abstract class AsyncTask<Params, Progress, Result> {
02     ......
03
04     private static class AsyncTaskResult<Data> {
05         final AsyncTask mTask;
06         final Data[] mData;
07
08         AsyncTaskResult(AsyncTask task, Data... data) {
09             mTask = task;
10             mData = data;
11         }
12     }
13
14     ......
15 }
```

它的成员变量mTask用来描述一个宿主异步任务，而成员变量mData用来描述这个宿主异步任务在执行过程中所产生的中间数据，或者在执行完成时所产生的结果数据。

从这里就可以看出，InternalHandler类的成员函数handleMessage分别调用了异步任务类AsyncTask的成员函数finish、onProgressUpdate和onCancelled来处理类型为MESSAGE_POST_RESULT、MESSAGE_POST_PROGRESS和MESSAGE_POST_CANCEL的消息，后面我们再分析它们的实现。

WorkerRunnable类实现了Callable<Result>接口，它的实现如下所示。

frameworks/base/core/java/android/os/AsyncTask.java

```
1 public abstract class AsyncTask<Params, Progress, Result> {
2     ......
3
4     private static abstract class WorkerRunnable<Params, Result> implements Callable<Result> {
5         Params[] mParams;
6     }
7
8     ......
9 }
```

它只有一个成员变量mParams，指向了一个类型为Params的数组，用来保存一个异步任务的输入数据。

了解了异步任务类AsyncTask的各个成员变量的含义之后，接下来我们就继续分析一个异步任务的创建过程。

异步任务类AsyncTask的构造函数的实现如下所示。

frameworks/base/core/java/android/os/AsyncTask.java

```
01 public abstract class AsyncTask<Params, Progress, Result> {
02     ......
03
04     public AsyncTask() {
05         mWorker = new WorkerRunnable<Params, Result>() {
06             public Result call() throws Exception {
07                 ......
08                 return doInBackground(mParams);
09             }
10         };
11
12         mFuture = new FutureTask<Result>(mWorker) {
13             @Override
14             protected void done() {
15                 Message message;
16                 Result result = null;
17
18                 try {
19                     result = get();
20                 } catch (InterruptedException e) {
21                     android.util.Log.w(LOG_TAG, e);
22                 } catch (ExecutionException e) {
23                     throw new RuntimeException("An error occured while executing doInBackground()",
24                         e.getCause());
25                 } catch (CancellationException e) {
26                     message = sHandler.obtainMessage(MESSAGE_POST_CANCEL,
27                         new AsyncTaskResult<Result>(AsyncTask.this, (Result[]) null));
28                     message.sendToTarget();
29                     return;
30                 } catch (Throwable t) {
31                     throw new RuntimeException("An error occured while executing "
32                         + "doInBackground()", t);
33                 }
34
35                 message = sHandler.obtainMessage(MESSAGE_POST_RESULT,
36                     new AsyncTaskResult<Result>(AsyncTask.this, result));
37                 message.sendToTarget();
38             }
39         };
40     }
41
42     ......
43 }
```

第5行到第10行代码首先创建了一个类型为WorkerRunnable<Params, Result>的对象，并且保存在异步任务类AsyncTask的成员变量mWorker中。它是用来描述一个即将要执行的工作任务的。

接着第12行到第39行代码将前面所创建的一个WorkerRunnable<Params, Result>对象封装成一个类型为FutureTask<Result>的对象，并且保存在异步任务类AsyncTask的成员变量mFuture中。它同样是用来描述一个即将执行的工作任务的。

从这里就可以看出，一个异步任务在创建完成之后，就会在内部获得一个WorkerRunnable<Params, Result>对象和一个FutureTask<Result>对象，它们共同描述了一个即将执行的工作任务。接下来，我们就继续分析这个工作任务的执行过程。

从CounterService组件的成员函数startCounter的实现可以知道，一个异步任务在创建完成之后，我们就可以调用它的成员函数execute来启动它。

异步任务类AsyncTask的成员函数execute的实现如下所示。

frameworks/base/core/java/android/os/AsyncTask.java

```
01 public abstract class AsyncTask<Params, Progress, Result> {
02     ......
03
04     public final AsyncTask<Params, Progress, Result> execute(Params... params) {
05         ......
06
07         mWorker.mParams = params;
08         sExecutor.execute(mFuture);
09
10         return this;
11     }
12
13     ......
14 }
```

第7行代码首先将参数params所描述的一个异步任务输入数据保存在异步任务类AsyncTask的成员变量mWorker中，接着第8行代码将异步任务类AsyncTask的成员变量mFuture所描述的一个工作任务，加入到异步任务类AsyncTask的静态成员变量sExecutor所描述的一个ThreadPoolExecutor线程池中去执行。

异步任务类AsyncTask的成员变量mFuture所描述的是一个抽象的工作任务，它在执行的过程中，会调用内部的一个Callable对象的成员函数call来执行一个具体的工作任务。从异步任务类AsyncTask的构造函数的实现可以知道，这个Callable对象指向了异步任务类AsyncTask的成员变量mWorker所描述的一个工作任务，因此，接下来就会调用WorkerRunnable<Params, Result>类的成员函数call来执行一个具体的任务。WorkerRunnable<Params, Result>类的成员函数call本身并不执行一个具体的任务，而是将这个具体的任务交给异步任务类AsyncTask的成员函数doInBackground来执行。

异步任务类AsyncTask的成员函数doInBackground是一个抽象方法，它的定义如下所示。

frameworks/base/core/java/android/os/AsyncTask.java

```
1 public abstract class AsyncTask<Params, Progress, Result> {
2     ......
3
4     protected abstract Result doInBackground(Params... params);
5
6     ......
7 }
```

AsyncTask子类通过重写这个抽象方法就可以执行一个自定义的异步任务。

从CounterService组件的成员函数startCounter的实现可以知道，一个异步任务在执行的过程中，如果需要执行与界面相关的操作，那么它就需要调用异步任务类AsyncTask的成员函数publishProgress，以便可以向应用程序主线程的消息队列发送一个消息。由于这个消息是在应用程序的主线程中处理的，因此，在处理这个消息的过程中，我们就可以执行一个与界面相关的操作了。

异步任务类AsyncTask的成员函数publishProgress的实现如下所示。

frameworks/base/core/java/android/os/AsyncTask.java

```
01 public abstract class AsyncTask<Params, Progress, Result> {
02     ......
03
04     protected final void publishProgress(Progress... values) {
05         sHandler.obtainMessage(MESSAGE_POST_PROGRESS,
06                 new AsyncTaskResult<Progress>(this, values)).sendToTarget();
07     }
08
09     ......
10 }
```

第5行和第6行代码首先将参数values所描述的一个异步任务执行过程数据封装成一个AsyncTaskResult<Progress>对象，然后将这个AsyncTaskResult<Progress>对象封装成一个类型为MESSAGE_POST_PROGRESS的消息，最后再通过异步任务类AsyncTask的静态成员变量sHandler所描述的一个Handler对象将这个消息发送到应用程序主线程的消息队列中。

通过异步任务类AsyncTask的静态成员变量sHandler发送的消息最终是由它的成员函数handleMessage来处理的。前面在分析这个成员函数的实现时提到，它最终将类型为MESSAGE_POST_PROGRESS的消息分发给异步任务类AsyncTask的成员函数onProgressUpdate来处理。

异步任务类AsyncTask的成员函数onProgressUpdate的实现如下所示。

frameworks/base/core/java/android/os/AsyncTask.java

```
1 public abstract class AsyncTask<Params, Progress, Result> {
2     ......
3
4     protected void onProgressUpdate(Progress... values) {
5     }
6
7     ......
8 }
```

AsyncTask子类通过重写这个成员函数就可以在一个异步任务的执行过程中执行一系列与界面相关的操作，因为这个成员函数是在应用程序的主线程中调用的。

从前面的分析可以知道，异步任务类AsyncTask所描述的一个工作任务是由一个类型为FutureTask<Result>的对象来表示的。当这个工作任务执行完成之后，与它所对应的一个FutureTask<Result>对象的成员函数done就会被调用。

FutureTask<Result>类的成员函数done首先会调用异步任务类AsyncTask的成员函数get来获得当前所执行的一个工作任务的结果数据，并且将这个结果数据封装成一个AsyncTaskResult<Result>对象，然后将这个AsyncTaskResult<Result>对象封装成一个类型为MESSAGE_POST_RESULT的消息，最后再通过异步任务类AsyncTask的静态成员变量sHandler所描述的一个Handler对象将这个消息发送到应用程序主线程的消息队列中。

异步任务类AsyncTask的成员函数get的实现如下所示。

frameworks/base/core/java/android/os/AsyncTask.java

```
1 public abstract class AsyncTask<Params, Progress, Result> {
2     ......
3
4     public final Result get() throws InterruptedException, ExecutionException {
5         return mFuture.get();
6     }
7
8     ......
9 }
```

第5行代码通过调用异步任务类AsyncTask的成员变量mFuture所描述的一个FutureTask<Result>对象的成员函数get，来获得它所描述的一个工作任务的执行结果数据，这个结果数据就对应于它内部的一个Callable对象的成员函数call的返回值；而后者的返回值又对应于一个AsyncTask子类的成员函数doInBackground的返回值。

从InternalHandler类的成员函数handleMessage的实现可以知道，类型为MESSAGE_POST_RESULT的消息最终是在异步任务类AsyncTask的成员函数finish中处理的。

异步任务类AsyncTask的成员函数finish的实现如下所示。

frameworks/base/core/java/android/os/AsyncTask.java

```
01 public abstract class AsyncTask<Params, Progress, Result> {
02     ......
03
04     private void finish(Result result) {
05         if (isCancelled()) result = null;
06         onPostExecute(result);
07         ......
08     }
09
10     ......
11 }
```

第5行的if语句首先检查当前正在处理的一个工作任务是否是因为被取消而结束执行的。如果是，那么就会将参数result的值设置为null。接下来第6行代码调用异步任务类AsyncTask的成员函数onPostExecute来继续处理类型为MESSAGE_POST_RESULT的消息。

异步任务类AsyncTask的成员函数onPostExecute的实现如下所示。

frameworks/base/core/java/android/os/AsyncTask.java

```
1 public abstract class AsyncTask<Params, Progress, Result> {
2     ......
3
4     protected void onPostExecute(Result result) {
5     }
6
7     ......
8 }
```

AsyncTask子类通过重写这个成员函数就可以在一个异步任务执行完成时执行一个与界面相关的操作，因为这个成员函数是在应用程序的主线程中调用的。

一个异步任务在执行的过程中是可以被取消的。在这种情况下，我们在获得它的执行结果数据时，就会得到一个类型为CancellationException的异常。这时候FutureTask<Result>类的成员函数done就会通过异步任务类AsyncTask的静态成员变量sHandler所描述的一个Handler对象，将一个类型为MESSAGE_POST_CANCEL的消息发送到应用程序主线程的消息队列中。

从InternalHandler类的成员函数handleMessage的实现可以知道，类型为MESSAGE_POST_CANCEL的消息最终是在异步任务类AsyncTask的成员函数onCancelled中处理的。

异步任务类AsyncTask的成员函数onCancelled的实现如下所示。

frameworks/base/core/java/android/os/AsyncTask.java

```
1 public abstract class AsyncTask<Params, Progress, Result> {
2     ......
3
4     protected void onCancelled() {
5     }
6
7     ......
8 }
```

AsyncTask子类通过重写这个成员函数就可以在一个异步任务被取消时执行一个与界面相关的操作，因为这个成员函数是在应用程序的主线程中调用的。

至此，与界面相关的Android应用程序子线程的消息循环模型就分析完成了，Android应用程序线程的三种消息循环模型也分析完成了。正确使用Android应用程序线程的消息循环模型，能够帮助我们提高应用程序主线程的界面事件响应能力，从而获得良好的用户体验。

第16章 Android应用程序的安装和显示过程

Android系统在启动过程中，会扫描系统中的特定目录，以便可以对保存在里面的应用程序进行安装。这是通过Package管理服务PackageManagerService来实现的。Package管理服务PackageManagerService在安装一个应用程序的过程中，主是完成两件事情。第一件事情是解析这个应用程序的配置文件AndroidManifest.xml，以便可以获得它的安装信息；第二件事情是为这个应用程序分配Linux用户ID和Linux用户组ID，以便它可以在系统中获得合适的运行权限。

Android系统启动过程的最后一步是启动一个Home应用程序，用来显示系统中已经安装了的应用程序。Android系统提供了一个默认的Home应用程序——Launcher。应用程序Launcher在启动过程中，首先会请求Package管理服务PackageManagerService返回系统中已经安装了的应用程序的信息，接着再分别将这些应用程序信息封装成一个快捷图标显示在系统的屏幕中，以便用户可以通过点击这些快捷图标来启动相应的应用程序。

接下来，我们首先分析Package管理服务PackageManagerService在系统启动时安装应用程序的过程，接着再分析应用程序Launcher显示系统中已经安装了的应用程序的过程。

16.1 应用程序的安装过程

Package管理服务PackageManagerService在安装一个应用程序的过程中，会对这个应用程序的配置文件AndroidManifest.xml进行解析，以便可以获得它的安装信息。一个应用程序可以配置的安装信息很多，其中，最重要的就是它的组件信息。从前面第7章到第10章的内容可以知道，一个应用程序组件只有在配置文件中被正确地配置之后，才可以被Activity管理服务ActivityManagerService正常地启动起来。

在前面的第10章中提到，Android系统中的每一个应用程序都有一个Linux用户ID。这些Linux用户ID就是由Package管理服务PackageManagerService来负责分配的。Package管理服务PackageManagerService在安装一个应用程序时，如果发现它没有与其他应用程序共享同一个Linux用户ID，那么就会为它分配一个唯一的Linux用户ID，以便它可以在系统中获得合适的运行权限。

一个应用程序除了拥有一个Linux用户ID之外，还可以拥有若干个Linux用户组ID，以便可以在系统中获得更多的资源访问权限，例如，读取联系人信息、使用摄像头、发送短信，以及拨打电话等权限。这些Linux用户组ID也是由Package管理服务PackageManagerService来负责分配的。Package管理服务PackageManagerService在安装一个应用程序时，如果发现它申请了一个特定的资源访问权限，那么就会为它分配一个相应的Linux用户组ID。

接下来在分析Android应用程序的安装过程中，我们主要关注它们的组件信息的解析过程，以及

它们的Linux用户ID和Linux用户组ID的分配过程。

从前面11.3小节的内容可以知道，System进程在启动时，会调用PackageManagerService类的静态成员函数main将系统的Package管理服务PackageManagerService启动起来。由于Package管理服务PackageManagerService在启动过程中，会对系统中的应用程序进行安装，因此，接下来我们就从PackageManagerService类的静态成员函数main开始分析应用程序的安装过程。

Package管理服务PackageManagerService安装应用程序的过程一共分为17个步骤，下面我们就详细分析每一个步骤。

第1步到第6步如图16–1所示。

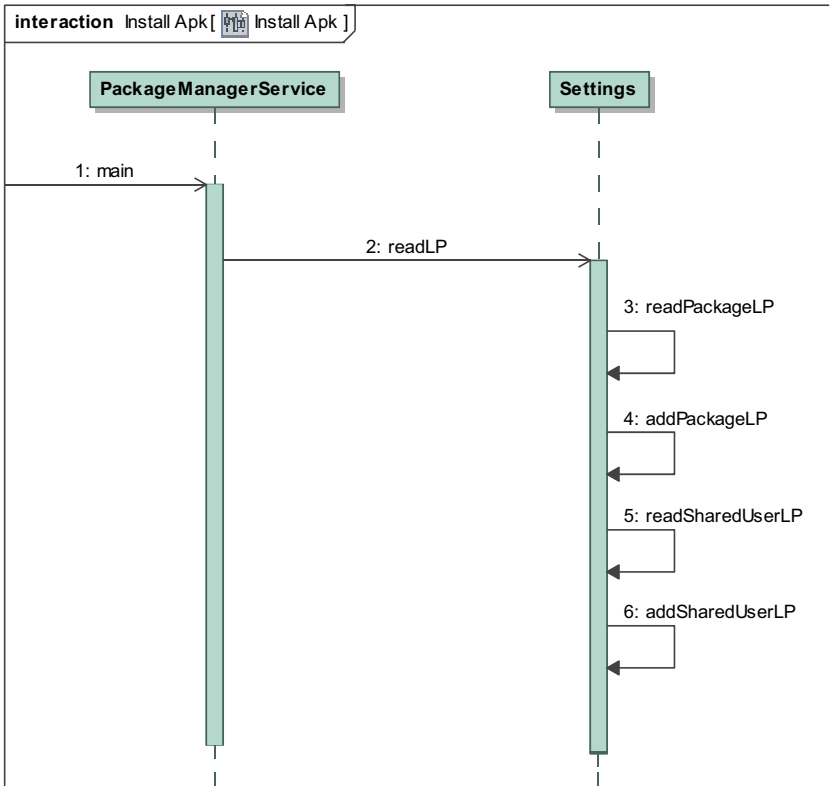


图16–1　应用程序的安装过程——Step 1至Step 6

Step 1：PackageManagerService.main

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     public static final IPackageManager main(Context context, boolean factoryTest) {
05         PackageManagerService m = new PackageManagerService(context, factoryTest);
06         ServiceManager.addService("package", m);
07         return m;
08     }
09
10     ......
11 }
```

第5行代码首先将Package管理服务PackageManagerService启动起来，接着第6行代码再将它注册到Service Manager中，这样系统中的其他组件就可以通过Service Manager来获得它的访问接口了。

Package管理服务PackageManagerService在启动过程中，会对系统中的特定目录进行扫描，以便可以对保存在里面的应用程序进行安装，相关的代码如下所示。

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     public PackageManagerService(Context context, boolean factoryTest) {
05         ......
06
07         mSettings = new Settings();
08         ......
09
10         synchronized (mInstallLock) {
11             synchronized (mPackages) {
12                 ......
13
14                 File dataDir = Environment.getDataDirectory();
15                 ......
16                 mDrmAppPrivateInstallDir = new File(dataDir, "app-private");
17                 ......
18
19                 mRestoredSettings = mSettings.readLP();
20                 ......
21
22                 mFrameworkDir = new File(Environment.getRootDirectory(), "framework");
23                 ......
24
25                 // Find base frameworks (resource packages without code).
26                 ......
27                 scanDirLI(mFrameworkDir, PackageParser.PARSE_IS_SYSTEM
28                     | PackageParser.PARSE_IS_SYSTEM_DIR,
29                     scanMode | SCAN_NO_DEX, 0);
30
31                 // Collect all system packages.
32                 mSystemAppDir = new File(Environment.getRootDirectory(), "app");
33                 ......
34                 scanDirLI(mSystemAppDir, PackageParser.PARSE_IS_SYSTEM
35                     | PackageParser.PARSE_IS_SYSTEM_DIR, scanMode, 0);
36
37                 // Collect all vendor packages.
38                 mVendorAppDir = new File("/vendor/app");
39                 ......
40                 scanDirLI(mVendorAppDir, PackageParser.PARSE_IS_SYSTEM
41                     | PackageParser.PARSE_IS_SYSTEM_DIR, scanMode, 0);
42                 ......
43
44                 mAppInstallDir = new File(dataDir, "app");
45                 ......
46                 scanDirLI(mAppInstallDir, 0, scanMode, 0);
47                 ......
48
49                 scanDirLI(mDrmAppPrivateInstallDir, PackageParser.PARSE_FORWARD_LOCK,
50                     scanMode, 0);
51                 ......
52
53                 final boolean regrantPermissions = mSettings.mInternalSdkPlatform
54                     != mSdkVersion;
55                 ......
56
57                 updatePermissionsLP(null, null, true, regrantPermissions, regrantPermissions);
58
59                 mSettings.writeLP();
```

```
60
61                  ......
62              }
63          }
64      }
65
66      ......
67  }
```

PackageManagerService类有一个类型为Settings的成员变量mSettings，它是用来管理应用程序的安装信息的。例如，用来管理Package管理服务PackageManagerService为应用程序分配的Linux用户ID和Linux用户组ID。

由于Android系统每次启动时，都会重新安装一遍系统中的应用程序，但是有些应用程序信息每次安装都是需要保持一致的，例如，应用程序的Linux用户ID；否则，应用程序每次在系统重新启动之后，表现可能都会不一致。因此，Package管理服务PackageManagerService每次在安装完成应用程序之后，都需要将它们的信息保存下来，以便下次安装时可以恢复回来。

恢复上一次的应用程序安装信息是通过调用PackageManagerService类的成员变量mSettings的成员函数readLP来实现的，如第19行代码所示。恢复了上一次的应用程序安装信息之后，接下来就开始扫描和安装保存在特定目录中的应用程序了。

第14行代码调用Environment类的静态成员函数getDataDirectory来获得系统的数据目录/data，并且保存在变量dataDir中；接下来第16行和第44行代码通过这个目录来获得另外两个目录mDrmAppPrivateInstallDir和mAppInstallDir，它们对应的系统路径分别是/data/app-private和/data/app。其中，/data/app-private目录保存的是受DRM保护的私有应用程序[1]，而/data/app目录保存的是由用户自己安装的应用程序。

第22行代码首先调用Environment类的静态成员函数getRootDirectory来获得Android系统目录/system，接下来再通过这个目录来获得另外一个目录mFrameworkDir，它对应的系统路径为/system/framework，里面保存的应用程序是资源型的。资源型的应用程序是用来打包资源文件的，它们不包含有执行代码。

第32行代码同样是首先调用Environment类的静态成员函数getRootDirectory来获得系统目录/system，接着再通过这个目录来获得另外一个目录mSystemAppDir，它对应的系统路径为/system/app，里面保存的是系统自带的应用程序。

第38行代码直接构建了一个目录mVendorAppDir，它对应的系统路径为/vendor/app，里面保存的是由设备厂商提供的应用程序。

这样，Package管理服务PackageManagerService就得到了五个目录，它们分别是/system/framework、/system/app、/vendor/app、/data/app和/data/app-private。接下来第27行、第34行、第40行、第46行和第49行代码就会分别调用PackageManagerService类的成员函数scanDirLI来安装保存在它们里面的应用程序。

安装完成保存在上述五个目录中的应用程序之后，接下来第57行代码再调用PackageManagerService类的成员函数updatePermissionsLP来为申请了特定的资源访问权限的应用程序分配相应的Linux用户组ID。

最后，第59行代码调用PackageManagerService的成员变量mSettings的成员函数writeLP将前面所获到得的应用程序安装信息保存在本地的一个配置文件中，以便下一次再安装这些应用程序时，可以将需要保持一致的应用程序信息恢复回来。

接下来，我们首先分析Settings类的成员函数readLP的实现，然后分析PackageManagerService类的成员函数scanDirLI和updatePermissionsLP的实现，最后分析Settings类的成员函数writeLP的实现。

1　DRM的全称为Digital Right Management，即数字权限管理技术，是用来保护数字内容版权的一种技术。

Step 2：Settings.readLP

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     private static final class Settings {
05         private final File mSettingsFilename;
06         private final File mBackupSettingsFilename;
07         ......
08
09         Settings() {
10             File dataDir = Environment.getDataDirectory();
11             File systemDir = new File(dataDir, "system");
12             ......
13
14             mSettingsFilename = new File(systemDir, "packages.xml");
15             mBackupSettingsFilename = new File(systemDir, "packages-backup.xml");
16             ......
17         }
18         ......
19
20         boolean readLP() {
21             FileInputStream str = null;
22             if (mBackupSettingsFilename.exists()) {
23                 try {
24                     str = new FileInputStream(mBackupSettingsFilename);
25                     ......
26                 } catch (java.io.IOException e) {
27                     // We'll try for the normal settings file.
28                 }
29             }
30
31             try {
32                 if (str == null) {
33                     ......
34                     str = new FileInputStream(mSettingsFilename);
35                 }
36                 XmlPullParser parser = Xml.newPullParser();
37                 parser.setInput(str, null);
38
39                 int type;
40                 ......
41
42                 int outerDepth = parser.getDepth();
43                 while ((type=parser.next()) != XmlPullParser.END_DOCUMENT
44                        && (type != XmlPullParser.END_TAG
45                                || parser.getDepth() > outerDepth)) {
46                     ......
47
48                     String tagName = parser.getName();
49                     if (tagName.equals("package")) {
50                         readPackageLP(parser);
51                     }
52                     ......
53                     else if (tagName.equals("shared-user")) {
54                         readSharedUserLP(parser);
55                     }
56                     ......
57                 }
58
59                 str.close();
60             } catch(XmlPullParserException e) {
```

```
61                  ......
62              } catch(java.io.IOException e) {
63                  ......
64              }
65
66              return true;
67          }
68
69          ......
70      }
71
72      ......
73  }
```

Settings类的成员变量mSettingsFilename和mBackupSettingsFilename分别指向了系统中的文件/data/system/packages.xml和/data/system/packages-backup.xml，这两个文件都是用来保存上一次的应用程序安装信息的，其中，后一个文件是用来备份前一个文件的。

第22行的if语句首先检查文件/data/system/packages-backup.xml是否存在。如果存在，那么接下来第24行代码就会以它的内容作为上一次的应用程序安装信息；否则，第34行代码就会以文件/data/system/packages.xml的内容作为上一次的应用程序安装信息。

由于文件/data/system/packages-backup.xml和文件/data/system/packages.xml的内容都是xml格式的，因此，第36行和第37行代码就创建一个XmlPullParser对象parser来解析保存在它们里面的应用程序安装信息。上一次安装应用程序时保存下来的信息比较多，不过我们主要关注与应用程序的Linux用户ID相关的信息，如第49行到第55行代码所示。

在文件/data/system/packages-backup.xml和文件/data/system/packages.xml中，每一个标签值为“package”的xml元素都是用来描述上一次安装的一个应用程序的信息的，第50行代码调用Settings类的成员函数readPackageLP来解析它的内容，以便可以获得上一次安装这个应用程序时所分配给它的Linux用户ID。

在文件/data/system/packages-backup.xml和文件/data/system/packages.xml中，每一个标签值为“shared-user”的xml元素都是用来描述上一次安装应用程序时所分配的一个共享Linux用户的，第54行代码调用Settings类的成员函数readSharedUserLP来解析它的内容，以便可以获得上一次安装应用程序时所分配的共享Linux用户ID。

接下来，我们首先分析Settings类的成员函数readPackageLP的实现，然后再分析Settings类的成员函数readSharedUserLP的实现。

Step 3：Settings. readPackageLP

```
frameworks/base/services/java/com/android/server/PackageManagerService.java
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     private static final class Settings {
05         ......
06
07         private final ArrayList<PendingPackage> mPendingPackages
08                 = new ArrayList<PendingPackage>();
09         ......
10
11         private void readPackageLP(XmlPullParser parser)
12                 throws XmlPullParserException, IOException {
13             String name = null;
14             ......
15             String idStr = null;
16             String sharedIdStr = null;
17             ......
```

```
18
19              try {
20                  name = parser.getAttributeValue(null, "name");
21                  ......
22                  idStr = parser.getAttributeValue(null, "userId");
23                  ......
24                  sharedIdStr = parser.getAttributeValue(null, "sharedUserId");
25                  ......
26
27                  int userId = idStr != null ? Integer.parseInt(idStr) : 0;
28                  ......
29
30                  if (name == null) {
31                      reportSettingsProblem(Log.WARN,
32                              "Error in package manager settings: <package> has no name at "
33                              + parser.getPositionDescription());
34                  }
35                  ......
36                  else if (userId > 0) {
37                      packageSetting = addPackageLP(name.intern(), realName, new File(codePathStr),
38                              new File(resourcePathStr), nativeLibraryPathStr, userId, versionCode,
39                              pkgFlags);
40                      ......
41                  } else if (sharedIdStr != null) {
42                      userId = sharedIdStr != null
43                              ? Integer.parseInt(sharedIdStr) : 0;
44                      if (userId > 0) {
45                          packageSetting = new PendingPackage(name.intern(), realName,
46                                  new File(codePathStr), new File(resourcePathStr),
47                                  nativeLibraryPathStr, userId, versionCode, pkgFlags);
48                          ......
49                          mPendingPackages.add((PendingPackage) packageSetting);
50                          ......
51                      }
52                      ......
53                  }
54                  ......
55              } catch (NumberFormatException e) {
56                  ......
57              }
58
59              ......
60          }
61
62          ......
63      }
64
65      ......
66  }
```

标签值等于“package”的xml元素有三个属性name、userId和sharedUserId，其中，属性name用来描述上一次安装的一个应用程序的Package名称；而属性userId和sharedUserId用来描述这个应用程序所使用的独立Linux用户ID和共享Linux用户ID。第20行、第22行和第24行代码分别将这三个属性的值解析出来，并且保存在变量name、idStr和sharedIdStr中。

上一次安装的每一个应用程序的Package名称都是必须存在的；否则，第31行到第33行代码就会调用PackageManagerService类的成员函数reportSettingsProblem来报告一个错误。

第36行的if语句检查Package名称等于name的应用程序在上一次安装时是否被分配了一个独立的Linux用户ID。如果是，那么接下来第37行到第39行代码就会调用Settings类的成员函数addPackageLP将这个应用程序上一次安装时被分配的Linux用户ID保留下来。

第41行的if语句检查前面所获得的变量sharedIdStr的值是否不等于null。如果不等于null，那么

就说明上一次安装Package名称等于name的应用程序时，Package管理服务PackageManagerService并没有给它分配一个独立的Linux用户ID，而是让它与其他的应用程序共享同一个Linux用户ID。在这种情况下，Package管理服务PackageManagerService不能马上为这个Package名称等于name的应用程序保留它上次所使用的Linux用户ID，因为这个Linux用户ID不属于它所有，换句话来说，就是这个Linux用户ID可能是一个无效的Linux用户ID。接下来第45行到第49行代码首先会创建一个PendingPackage对象，接着再将这个PendingPackage对象保存在Settings类的成员变量mPendingPackages中，用来描述一个Linux用户ID还未确定的应用程序。在接下来的Step 6中，我们就会看到，等到Package管理服务PackageManagerService解析完成上一次的应用程序安装信息中的共享Linux用户信息之后，才能确认保存在Settings类的成员变量mPendingPackages中的应用程序上次所使用的一个共享Linux用户ID是否还是有效的。如果还是有效，那么Package管理服务PackageManagerService才会为它们保留它们上一次所使用的Linux用户ID。

接下来，我们继续分析Settings类的成员函数addPackageLP的实现，以便可以了解它是如何为上一次安装的一个应用程序保留一个Linux用户ID的。

Step 4：Settings. addPackageLP

```
frameworks/base/services/java/com/android/server/PackageManagerService.java
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     private static final class Settings {
05         ......
06
07         private final HashMap<String, PackageSetting> mPackages =
08                 new HashMap<String, PackageSetting>();
09         ......
10
11         PackageSetting addPackageLP(String name, String realName, File codePath, File resourcePath,
12                 String nativeLibraryPathString, int uid, int vc, int pkgFlags) {
13             PackageSetting p = mPackages.get(name);
14             if (p != null) {
15                 if (p.userId == uid) {
16                     return p;
17                 }
18                 ......
19                 return null;
20             }
21             p = new PackageSetting(name, realName, codePath, resourcePath, nativeLibraryPathString,
22                     vc, pkgFlags);
23             p.userId = uid;
24             if (addUserIdLP(uid, p, name)) {
25                 mPackages.put(name, p);
26                 return p;
27             }
28             return null;
29         }
30
31         ......
32     }
33
34     ......
35 }
```

在Package管理服务PackageManagerService中，每一个应用程序的安装信息都是使用一个PackageSetting对象来描述的。这些PackageSetting对象被保存在Settings类的成员变量mPackages所描述的一个HashMap中，并且是以它们所描述的应用程序的Package名称为关键字的。

从前面的Step 3可以知道，参数name用来描述一个应用程序的Package名称，第13行和第14行代码首先在Settings类的成员变量mPackages中检查是否已经存在一个与它所对应的PackageSetting对象。如果存在，那么接下来第15行的if语句再检查这个PackageSetting对象的成员变量userId是否等于参数uid的值。如果等于，那么就说明Package管理服务PackageManagerService已经为Package名称等于name的应用程序分配过所要求的一个Linux用户ID了，因此，接下来第16行代码就直接返回了。

如果Settings类的成员变量mPackages中不存在与参数name所对应的一个PackageSetting对象，即第14行的if语句为false，那么接下来第21行到第27行代码就会为Package名称等于name的应用程序分配参数uid所描述的一个Linux用户ID。

第21行和第22行代码首先创建了一个PackageSetting对象p，用来描述Package名称等于name的应用程序的安装信息；接着第23行代码将这个PackageSetting对象p的成员变量userId的值设置为参数uid的值，以便可以表示Package名称等于name的应用程序所使用的Linux用户ID的值为uid；最后第24行代码调用Settings类的成员函数addUserIdLP在系统中保留值为uid的Linux用户ID。如果保留成功，那么第25行代码还会将前面所创建的一个PackageSetting对象p保存在Settings类的成员变量mPackages中，以便可以表示Package管理服务PackageManagerService已经为Package名称等于name的应用程序分配过Linux用户ID了。

接下来，我们继续分析Settings类的成员函数addUserIdLP是如何在系统中保留一个指定的Linux用户ID的，它的实现如下所示。

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     private static final class Settings {
05         ......
06
07         private final ArrayList<Object> mUserIds = new ArrayList<Object>();
08         private final SparseArray<Object> mOtherUserIds =
09                 new SparseArray<Object>();
10         ......
11
12         private boolean addUserIdLP(int uid, Object obj, Object name) {
13             if (uid >= FIRST_APPLICATION_UID + MAX_APPLICATION_UIDS) {
14                 return false;
15             }
16
17             if (uid >= FIRST_APPLICATION_UID) {
18                 int N = mUserIds.size();
19                 final int index = uid - FIRST_APPLICATION_UID;
20                 while (index >= N) {
21                     mUserIds.add(null);
22                     N++;
23                 }
24                 if (mUserIds.get(index) != null) {
25                     ......
26                     return false;
27                 }
28                 mUserIds.set(index, obj);
29             } else {
30                 if (mOtherUserIds.get(uid) != null) {
31                     ......
32                     return false;
33                 }
34                 mOtherUserIds.put(uid, obj);
35             }
36             return true;
```

```
37          }
38
39          ......
40      }
41
42      ......
43  }
```

在Android系统中，大于或者等于FIRST_APPLICATION_UID并且小于（FIRST_APPLICATION_UID + MAX_APPLICATION_UIDS）的Linux用户ID是保留给应用程序使用的，而小于FIRST_APPLICATION_UID的Linux用户ID是保留给特权用户使用的。FIRST_APPLICATION_UID和MAX_APPLICATION_UIDS的值分别定义为10000和1000，从这里就可以看出，系统最多可以分配1000个Linux用户ID给应用程序使用。虽然小于FIRST_APPLICATION_UID的Linux用户ID不能作为应用程序的Linux用户ID，但是它们却可以以共享的方式被应用程序使用。例如，如果一个应用程序想要修改系统的时间，那么它就可以申请与名称为“android.uid.system”的特权用户共享同一个Linux用户ID（1000），即在配置文件中将它的android:sharedUserId属性值设置为“android.uid.system”。

Settings类的成员变量mUserIds是一个类型为ArrayList的列表，用来维护那些已经分配给应用程序使用的Linux用户ID，即大于或者等于FIRST_APPLICATION_UID的Linux用户ID。如果保存在这个列表中的第index个位置的一个Object对象的值不等于null，那么就说明值为（FIRST_APPLICATION_UID + index）的Linux用户ID已经被分配了。

Settings类的另外一个成员变量mOtherUserIds是一个类型为SparseArray的稀疏数组，用来维护那些已经分配给特权用户使用的Linux用户ID，即小于FIRST_APPLICATION_UID的Linux用户ID。如果一个小于FIRST_APPLICATION_UID的Linux用户ID已经被分配了，那么它在这个稀疏数组中所对应的一个Object对象的值就不等于null。

注意

虽然保存在Settings类的成员变量mUserIds和mOtherUserIds中的对象的类型为Object，但是它们实际上指向的要么是一个PackageSetting对象，要么是一个SharedUserSetting对象。其中，PackageSetting对象所描述的Linux用户ID是独立的，而SharedUserSetting对象所描述的Linux用户ID是共享的。

第13行的if语句首先检查要保留的Linux用户ID的值是否大于或者等于（FIRST_APPLICATION_UID + MAX_APPLICATION_UIDS）。如果是，那么就说明要保留的是一个非法的Linux用户ID，因此，接下来第14行代码就直接返回一个false值给调用者。

第17行的if语句接着检查要保留的Linux用户ID的值是否大于或者等于FIRST_APPLICATION_UID。如果是，那么接下来第18行到第28行代码就会将它添加到Setting类的成员变量mUserIds中；否则，第30行到第34行代码就会将它添加到Setting类的成员变量mOtherUserIds中。

这一步执行完成之后，返回到前面的Step 2中，即Settings类的成员函数readLP中，接下来就会调用Settings类的成员函数readSharedUserLP来解析上一次的应用程序安装信息中的共享Linux用户信息。

Step 5：Settings. readSharedUserLP

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     private static final class Settings {
05         ......
```

```
06
07          private void readSharedUserLP(XmlPullParser parser)
08                  throws XmlPullParserException, IOException {
09              String name = null;
10              String idStr = null;
11              int pkgFlags = 0;
12              SharedUserSetting su = null;
13              try {
14                  name = parser.getAttributeValue(null, "name");
15                  idStr = parser.getAttributeValue(null, "userId");
16                  int userId = idStr != null ? Integer.parseInt(idStr) : 0;
17                  if ("true".equals(parser.getAttributeValue(null, "system"))) {
18                      pkgFlags |= ApplicationInfo.FLAG_SYSTEM;
19                  }
20                  if (name == null) {
21                      ......
22                  } else if (userId == 0) {
23                      ......
24                  } else {
25                      if ((su=addSharedUserLP(name.intern(), userId, pkgFlags)) == null) {
26                          reportSettingsProblem(Log.ERROR,
27                                  "Occurred while parsing settings at "
28                                  + parser.getPositionDescription());
29                      }
30                  }
31              } catch (NumberFormatException e) {
32                  ......
33              };
34
35              ......
36          }
37
38          ......
39      }
40
41      ......
42  }
```

在文件/data/system/packages-backup.xml或者文件/data/system/packages.xml中，每一个标签值等于“shared-user”的xml元素都是用来描述上一次安装应用程序时所分配的一个共享Linux用户的。这个xml元素有三个属性name、userId和system，其中，属性name和userId描述一个共享Linux用户的名称和ID值；而属性system用来描述这个共享Linux用户ID是分配给一个系统类型的应用程序使用的，还是分配给一个用户类型的应用程序使用的。

第14行和第15行代码分别将上一次安装应用程序时所分配的一个共享Linux用户的名称和ID值取出来，并且保存在变量name和idStr中；接着第16行代码再将变量idStr的值转换为一个整数，并且保存在变量userId中。

第17行的if语句检查值等于userId的共享Linux用户ID是否是分配给一个系统类型的应用程序使用的。如果是，那么接下来第18行代码就会将变量pkgFlags的第ApplicationInfo.FLAG_SYSTEM位的值设置为1。

由于标签值等于“shared-user”的xml元素的属性name和userId是要求存在的，因此，接下来第25行代码就会调用Settings类的成员函数addSharedUserLP在系统中为名称等于name的共享Linux用户保留一个值为userId的Linux用户ID。

Step 6：Settings. addSharedUserLP

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     private static final class Settings {
05         ......
06
07         private final HashMap<String, SharedUserSetting> mSharedUsers =
08                 new HashMap<String, SharedUserSetting>();
09         ......
10
11         SharedUserSetting addSharedUserLP(String name, int uid, int pkgFlags) {
12             SharedUserSetting s = mSharedUsers.get(name);
13             if (s != null) {
14                 if (s.userId == uid) {
15                     return s;
16                 }
17                 ......
18                 return null;
19             }
20             s = new SharedUserSetting(name, pkgFlags);
21             s.userId = uid;
22             if (addUserIdLP(uid, s, name)) {
23                 mSharedUsers.put(name, s);
24                 return s;
25             }
26             return null;
27         }
28
29         ......
30     }
31
32     ......
33 }
```

在Package管理服务PackageManagerService中，每一个共享Linux用户都是使用一个SharedUserSetting对象来描述的。这些SharedUserSetting对象被保存在Settings类的成员变量mSharedUsers所描述的一个HashMap中，并且是以它们所描述的共享Linux用户的名称为关键字的。

从前面的Step 5可以知道，参数name用来描述一个共享Linux用户的名称，第12行和第13行代码首先在Settings类的成员变量mSharedUsers中检查是否已经存在一个与它所对应的SharedUserSetting对象。如果存在，那么接下来第14行的if语句再检查这个SharedUserSetting对象的成员变量userId的值是否等于参数uid的值。如果等于，那么就说明Package管理服务PackageManagerService已经为名称等于name的共享Linux用户分配过所要求的一个Linux用户ID了，因此，接下来第15行代码就直接返回了。

如果Settings类的成员变量mSharedUsers中不存在与参数name所对应的一个SharedUserSetting对象，即第13行的if语句为false，那么接下来第20行到第25行代码就会为名称等于name的共享Linux用户分配参数uid所描述的一个Linux用户ID。

第20行代码首先创建了一个SharedUserSetting对象s，用来描述名称等于name的共享Linux用户；接着第21行代码再将这个SharedUserSetting对象s的成员变量userId的值设置为参数uid的值，以便可以表示名称等于name的共享Linux用户所使用的Linux用户ID的值为uid；最后第22行代码调用Settings类的成员函数addUserIdLP在系统中保留一个值为uid的Linux用户ID。如果保留成功，那么第23行代码还会将前面所创建的一个SharedUserSetting对象s保存在Settings类的成员变量mSharedUsers中，以便可以表示Package管理服务PackageManagerService已经为名称等于name的共享Linux用户分配过Linux用户ID了。

注意

第22行代码在调用Settings类的成员函数addUserIdLP在系统中保留一个Linux用户ID时，传进去的第二参数s的类型为SharedUserSetting。这时候在Settings类的成员变量mUserIds中，与参数uid所对应的一个Object对象的类型就为SharedUserSetting。这样Package管理服务PackageManagerService就可以知道值为uid的Linux用户ID是分配给一个共享Linux用户使用的。

这一步执行完成之后，再次返回到前面的Step 2中，即Settings类的成员函数readLP中，这时候Package管理服务PackageManagerService就获得上一次安装应用程序时所分配的共享Linux用户信息了。

在前面的Step 3中提到，如果上一次安装的一个应用程序指定了要与其他应用程序共享同一个Linux用户ID，那么Package管理服务PackageManagerService需要在解析完成上一次的应用程序安装信息中的共享Linux用户信息之后，再为它们保留它们上一次所使用的Linux用户ID。这些未保留Linux用户ID的应用程序分别被封装成一个PendingPackage对象，并且保存在Settings类的成员变量mPendingPackages中。

现在既然Package管理服务PackageManagerService已经获得了上一次安装应用程序时所分配的共享Linux用户信息，接下来我们就继续分析Settings类的成员函数readLP是如何为保存在Settings类的成员变量mPendingPackages中的应用程序保留它们上一次所使用的Linux用户ID的。

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     private static final class Settings {
05         ......
06
07         boolean readLP() {
08             ......
09
10             int N = mPendingPackages.size();
11             for (int i=0; i<N; i++) {
12                 final PendingPackage pp = mPendingPackages.get(i);
13                 Object idObj = getUserIdLP(pp.sharedId);
14                 if (idObj != null && idObj instanceof SharedUserSetting) {
15                     PackageSetting p = getPackageLP(pp.name, null, pp.realName,
16                             (SharedUserSetting) idObj, pp.codePath, pp.resourcePath,
17                             pp.nativeLibraryPathString, pp.versionCode, pp.pkgFlags, true, true);
18                     ......
19                     p.copyFrom(pp);
20                 }
21                 ......
22             }
23             mPendingPackages.clear();
24
25             ......
26         }
27
28         ......
29     }
30
31     ......
32 }
```

第11行到第22行的for循环依次遍历保存在Settings类的成员变量mPendingPackages中的每一个PendingPackage对象，以便可以为它们所描述的应用程序保留上次安装时所使用的Linux用户ID。

对于保存在Settings类的成员变量mPendingPackages中的每一个PendingPackage对象pp来说，如果在Settings类的成员变量mUserIds或者mOtherUserIds中存在一个与它的成员变量userId所对应的Object对象，并且这个Object对象的实际类型为SharedUserSetting，即第14行的if语句为true，那么就说明PendingPackage对象pp所描述的一个应用程序上一次所使用的一个Linux用户ID是有效的。因此，接下来第15行到第19行代码就会调用Settings类的成员函数getPackageLP来为它保留它上一次安装时所使用的一个Linux用户ID，即为它创建一个PackageSetting对象，并且保存在Settings类的成员变量mPackages中。在接下来的Step 12中，我们再分析Settings类的成员函数getPackageLP的实现。

这一步执行完成之后，返回到前面的Step 1中，即PackageManagerService类的构造函数中，这时候Package管理服务PackageManagerService就将上一次的应用程序安装信息恢复完成了。接下来第7步到第13步就会继续调用PackageManagerService类的成员函数scanDirLI来安装保存在目录/system/framework、/system/app、/vendor/app、/data/app和/data/app-private中的应用程序，这个过程如图16-2所示。

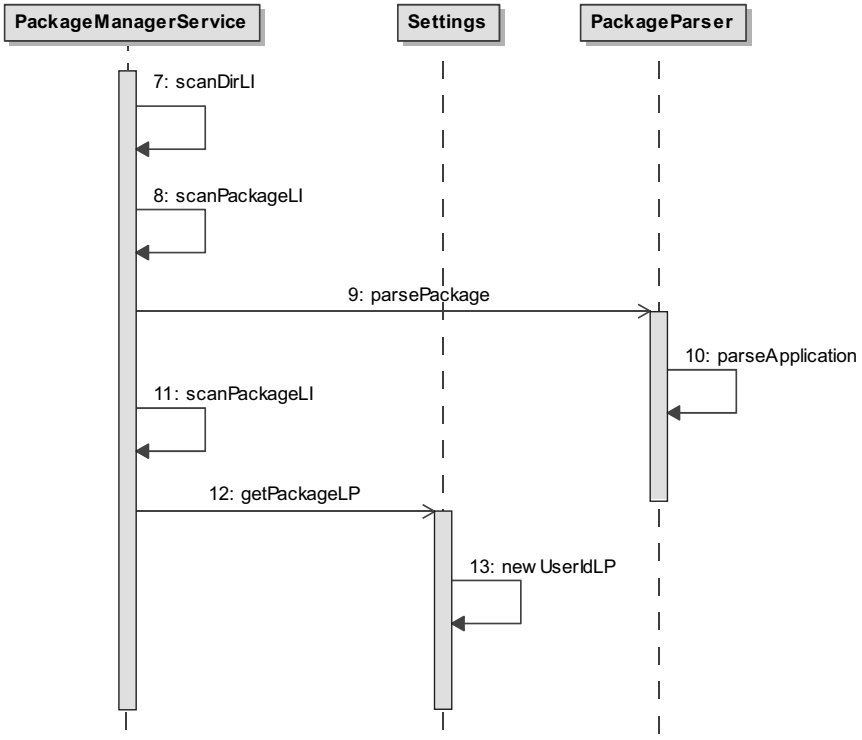


图16-2　应用程序的安装过程——Step 7至Step 13

Step 7：PackageManagerService.scanDirLI

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     private void scanDirLI(File dir, int flags, int scanMode, long currentTime) {
05         String[] files = dir.list();
06         ......
07
08         int i;
09         for (i=0; i<files.length; i++) {
10             File file = new File(dir, files[i]);
11             if (!isPackageFilename(files[i])) {
12                 // Ignore entries which are not apk's
```

```
13                  continue;
14              }
15              PackageParser.Package pkg = scanPackageLI(file,
16                  flags|PackageParser.PARSE_MUST_BE_APK, scanMode, currentTime);
17              // Don't mess around with apps in system partition.
18              if (pkg == null && (flags & PackageParser.PARSE_IS_SYSTEM) == 0 &&
19                  mLastScanError == PackageManager.INSTALL_FAILED_INVALID_APK) {
20                      // Delete the apk
21                      Slog.w(TAG, "Cleaning up failed install of " + file);
22                      file.delete();
23              }
24          }
25      }
26
27      ......
28  }
```

第9行到第24行的for循环依次检查保存在参数dir所描述的一个目录中的每一个文件。如果这些文件的名称是以“.apk”为后缀的，即第11行的if语句为true，那么接下来第15行和第16行代码就会调用PackageManagerService类的成员函数scanPackageLI来对它们进行解析。每解析完成一个apk文件之后，PackageManagerService类的成员函数scanPackageLI就会返回一个Package对象pkg给调用者。

如果PackageManagerService类的成员函数scanPackageLI发现它所解析的apk文件并不是一个真正的应用程序文件，那么它就会将PackageManagerService类的成员变量mLastScanError的值设置为PackageManager.INSTALL_FAILED_INVALID_APK，并且返回一个值为null的Package对象pkg给调用者。在这种情况下，如果参数flags的第PackageParser.PARSE_IS_SYSTEM位的值等于0，即参数dir所描述的一个目录不是一个系统目录，那么第22行代码就会将刚才所解析的文件删除。从前面的Step 1可以知道，目录/data/app和/data/app-private均不是系统目录，因此，当保存在它里面的一个apk文件不是一个真正的应用程序文件时，这个文件就会被删除。

接下来，我们继续分析PackageManagerService类的成员函数scanPackageLI的实现，以便可以了解它是如何解析一个应用程序文件的。

Step 8：PackageManagerService.scanPackageLI

```
frameworks/base/services/java/com/android/server/PackageManagerService.java
01 class PackageManagerService extends IPackageManager.Stub {
02      ......
03
04      private PackageParser.Package scanPackageLI(File scanFile,
05              int parseFlags, int scanMode, long currentTime) {
06          ......
07
08          String scanPath = scanFile.getPath();
09          ......
10          PackageParser pp = new PackageParser(scanPath);
11          ......
12
13          final PackageParser.Package pkg = pp.parsePackage(scanFile,
14              scanPath, mMetrics, parseFlags);
15          ......
16
17          return scanPackageLI(pkg, parseFlags, scanMode | SCAN_UPDATE_SIGNATURE, currentTime);
18      }
19
20      ......
21  }
```

第8行的代码首先得到要解析的应用程序文件所对应的路径名称，接着第10行到第14行代码再以这个路径名称为参数来创建一个PackageParser对象，并且调用这个PackageParser对象的成员函数parsePackage来对它所描述的一个应用程序文件进行解析。

PackageParser类的成员函数parsePackage成功地解析完成一个应用程序文件之后，就会返回一个Package对象给PackageManagerService类的成员函数scanPackageLI，后者接着会调用PackageManagerService类的另外一个重载版本的成员函数scanPackageLI来对这个Package对象所描述的一个应用程序文件进行安装，以便可以获得它的组件信息，以及为它分配的Linux用户ID等，如第17行代码所示。

接下来，我们首先分析PackageParser类的成员函数parsePackage的实现，然后分析PackageManagerService类的另外一个重载版本的成员函数scanPackageLI的实现。

Step 9：PackageParser.parsePackage

frameworks/base/core/java/android/content/pm/PackageParser.java

```
01 public class PackageParser {
02     ......
03
04     public Package parsePackage(File sourceFile, String destCodePath,
05             DisplayMetrics metrics, int flags) {
06         ......
07
08         mArchiveSourcePath = sourceFile.getPath();
09         ......
10
11         XmlResourceParser parser = null;
12         AssetManager assmgr = null;
13         ......
14         try {
15             assmgr = new AssetManager();
16             int cookie = assmgr.addAssetPath(mArchiveSourcePath);
17             if(cookie != 0) {
18                 parser = assmgr.openXmlResourceParser(cookie, "AndroidManifest.xml");
19                 ......
20             }
21             ......
22         } catch (Exception e) {
23             ......
24         }
25         ......
26
27         String[] errorText = new String[1];
28         Package pkg = null;
29         ......
30         try {
31             // XXXX todo: need to figure out correct configuration.
32             Resources res = new Resources(assmgr, metrics, null);
33             pkg = parsePackage(res, parser, flags, errorText);
34         } catch (Exception e) {
35             ......
36         }
37         ......
38
39         parser.close();
40         assmgr.close();
41         ......
42
```

```
43          return pkg;
44      }
45
46      ......
47  }
```

第8行代码首先将参数sourceFile所描述的一个应用程序文件的路径名称保存在PackageParser类的成员变量mArchiveSourcePath中，接着第11行到第36行的代码再对这个应用程序进行解析。

Android应用程序文件是一个以“.apk”为后缀名的归档文件，它里面包含了一个配置文件AndroidManifest.xml。通过解析这个配置文件，我们就可以获得一个应用程序的所有信息。因此，解析一个Android应用程序文件的过程其实就是解析包含在它里面的一个配置文件AndroidManifest.xml的过程。

第15行到第20行代码首先获得保存在要解析的Android应用程序文件中的配置文件AndroidManifest.xml，接着第32行和第33行代码调用PackageParser类的另外一个重载版本的成员函数parsePackage来对这个AndroidManifest.xml文件进行解析，以便可以获得对应的Android应用程序文件的信息。

PackageParser类的另外一个重载版本的成员函数parsePackage的实现如下所示。

frameworks/base/core/java/android/content/pm/PackageParser.java

```
01 public class PackageParser {
02     ......
03
04     private Package parsePackage(
05             Resources res, XmlResourceParser parser, int flags, String[] outError)
06             throws XmlPullParserException, IOException {
07         AttributeSet attrs = parser;
08         ......
09
10         String pkgName = parsePackageName(parser, attrs, flags, outError);
11         ......
12
13         final Package pkg = new Package(pkgName);
14         ......
15
16         TypedArray sa = res.obtainAttributes(attrs,
17                 com.android.internal.R.styleable.AndroidManifest);
18         ......
19
20         String str = sa.getNonConfigurationString(
21                 com.android.internal.R.styleable.AndroidManifest_sharedUserId, 0);
22         if (str != null && str.length() > 0) {
23             ......
24             pkg.mSharedUserId = str.intern();
25             ......
26         }
27         sa.recycle();
28         ......
29
30         int type;
31         ......
32
33         while ((type=parser.next()) != parser.END_DOCUMENT
34             && (type != parser.END_TAG || parser.getDepth() > outerDepth)) {
35             ......
36
37             String tagName = parser.getName();
38             if (tagName.equals("application")) {
39                 ......
```

```
40
41                  if (!parseApplication(pkg, res, parser, attrs, flags, outError)) {
42                      return null;
43                  }
44              }
45              ......
46              else if (tagName.equals("uses-permission")) {
47                  sa = res.obtainAttributes(attrs,
48                          com.android.internal.R.styleable.AndroidManifestUsesPermission);
49
50                  // Note: don't allow this value to be a reference to a resource
51                  // that may change.
52                  String name = sa.getNonResourceString(
53                          com.android.internal.R.styleable.AndroidManifestUsesPermission_name);
54
55                  sa.recycle();
56
57                  if (name != null && !pkg.requestedPermissions.contains(name)) {
58                      pkg.requestedPermissions.add(name.intern());
59                  }
60                  ......
61              }
62              ......
63          }
64          ......
65
66          return pkg;
67      }
68
69      ......
70  }
```

参数res和parser分别指向了一个Resources对象和一个XmlResourceParser对象，它们均是用来访问一个xml文件的内容的。从前面的调用过程可以知道，这个xml文件就是我们所需要解析的应用程序配置文件AndroidManifest.xml。

第10行代码首先调用PackageParser类的成员函数parsePackageName来获得要解析的应用程序的Package名称，接着第13行代码再以这个Package名称为参数来创建一个Package对象pkg，以便可以用来保存即将要解析的应用程序的配置信息。

我们可以在一个AndroidManifest.xml文件中配置一个应用程序的很多信息，不过这里我们只关心与共享Linux用户ID、Linux用户组ID和应用程序组件相关的配置信息，这些配置信息分别保存在manifest、uses-permission和application标签中[1]。

第16行到第21行代码用来解析manifest标签中的android:sharedUserId属性。如果我们设置了这个属性，那么就表示正在解析的应用程序要与其他应用程序共享同一个Linux用户ID。在这种情况下，第24行代码就会将这个属性值提取出来，并且将它保存在前面所创建的一个Package对象pkg的成员变量mSharedUserId中。

第33行到第63行的while循环用来解析uses-permission和application标签，这两个标签均为manifest标签里面的一个子标签。

一个uses-permission标签对应一个资源访问权限。在后面的Step 15中，我们就会看到，一个资源访问权限又是与一个Linux用户组ID相对应的，即如果一个应用程序申请了某一个资源访问权限，那么它就会获得一个对应的Linux用户组ID。

1　应用程序配置文件AndroidManifest.xml的完整结构可以参考官方文档：http://developer.android.com/guide/topics/manifest/manifest-intro.html。

一个应用程序可以同时申请多个资源访问权限，即可以在它的配置文件中设置多个uses-permission标签。这些uses-permission标签有一个属性name，用来描述它所对应的一个资源访问权限的名称。第47行到第59行代码将正在解析的应用程序所申请的资源访问权限的名称保存在前面所创建的一个Package对象pkg的成员变量requestedPermissions中。

每一个AndroidManifest.xml文件都必须有一个application标签，用来描述与应用程序组件相关的信息。第41行代码调用PackageParser类的成员函数parseApplication来解析这个标签的内容，以便可以获得正在解析的应用程序的组件配置信息。

接下来，我们就继续分析PackageParser类的成员函数parseApplication的实现。

Step 10：PackageParser.parseApplication

frameworks/base/core/java/android/content/pm/PackageParser.java

```
01 public class PackageParser {
02     ......
03
04     private boolean parseApplication(Package owner, Resources res,
05             XmlPullParser parser, AttributeSet attrs, int flags, String[] outError)
06             throws XmlPullParserException, IOException {
07         ......
08
09         int type;
10         while ((type=parser.next()) != parser.END_DOCUMENT
11             && (type != parser.END_TAG || parser.getDepth() > innerDepth)) {
12             ......
13
14             String tagName = parser.getName();
15             if (tagName.equals("activity")) {
16                 Activity a = parseActivity(owner, res, parser, attrs, flags, outError, false);
17                 ......
18
19                 owner.activities.add(a);
20             } else if (tagName.equals("receiver")) {
21                 Activity a = parseActivity(owner, res, parser, attrs, flags, outError, true);
22                 ......
23
24                 owner.receivers.add(a);
25             } else if (tagName.equals("service")) {
26                 Service s = parseService(owner, res, parser, attrs, flags, outError);
27                 ......
28
29                 owner.services.add(s);
30             } else if (tagName.equals("provider")) {
31                 Provider p = parseProvider(owner, res, parser, attrs, flags, outError);
32                 ......
33
34                 owner.providers.add(p);
35             }
36            ......
37         }
38
39         return true;
40     }
41
42     ......
43 }
```

在应用程序配置文件AndroidManifest.xml中，分别使用activity、receiver、service和provider标签来描述一个应用程序的Activity、Broadcast Receiver、Service和Content Provider组件的配置信息，它们均

是application标签里面的一个子标签。

第10行到第37行的while循环分别对activity、receiver、service和provider标签进行解析，以便可以获得正在解析的应用程序的Activity、Broadcast Receiver、Service和Content Provider组件的配置信息，这些组件的配置信息分别保存在参数owner所描述的一个Package对象的成员变量activities、receivers、services和providers中。

这一步执行完成之后，返回到前面的Step 8中，即PackageManagerService类的成员函数scanPackageLI中，这时候Package管理服务PackageManagerService就解析完成一个应用程序了。接下来它再调用PackageManagerService类的另外一个重载版本的成员函数scanPackageLI，以便可以获得前面所解析的应用程序的组件配置信息，以及为这个应用程序分配Linux用户ID。

Step 11：PackageManagerService.scanPackageLI

在分析这个函数之前，我们首先解释PackageManagerService类的五个成员变量mPackages、mActivities、mReceivers、mServices和mProvidersByComponent的作用，它们的定义如下所示：

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     // Keys are String (package name), values are Package.  This also serves
05     // as the lock for the global state.  Methods that must be called with
06     // this lock held have the prefix "LP".
07     final HashMap<String, PackageParser.Package> mPackages =
08         new HashMap<String, PackageParser.Package>();
09     ......
10
11     // All available activities, for your resolving pleasure.
12     final ActivityIntentResolver mActivities =
13         new ActivityIntentResolver();
14
15     // All available receivers, for your resolving pleasure.
16     final ActivityIntentResolver mReceivers =
17         new ActivityIntentResolver();
18
19     // All available services, for your resolving pleasure.
20     final ServiceIntentResolver mServices = new ServiceIntentResolver();
21
22     // Keys are String (provider class name), values are Provider.
23     final HashMap<ComponentName, PackageParser.Provider> mProvidersByComponent =
24         new HashMap<ComponentName, PackageParser.Provider>();
25
26     ......
27 }
```

系统中所有已经安装了的应用程序都是使用一个Package对象来描述的。这些Package对象保存在PackageManagerService类的成员变量mPackages所描述的一个HashMap中，并且是以这些Package对象所描述的应用程序的Package名称为关键字的。

系统中每一个已经安装了的应用程序都包含了一系列的Activity、Broadcast Receiver、Service和Content Provider组件，这些组件的配置信息分别保存在PackageManagerService类的成员变量mActivities、mReceivers、mServices和mProvidersByComponent中。

接下来，我们就开始分析PackageManagerService类的另外一个重载版本的成员函数scanPackageLI的实现，如下所示。

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     private PackageParser.Package scanPackageLI(PackageParser.Package pkg,
05             int parseFlags, int scanMode, long currentTime) {
06         ......
07
08         SharedUserSetting suid = null;
09         PackageSetting pkgSetting = null;
10         ......
11
12         synchronized (mPackages) {
13             ......
14
15             if (pkg.mSharedUserId != null) {
16                 suid = mSettings.getSharedUserLP(pkg.mSharedUserId,
17                         pkg.applicationInfo.flags, true);
18                 if (suid == null) {
19                     ......
20                     return null;
21                 }
22                 ......
23             }
24
25             pkgSetting = mSettings.getPackageLP(pkg, origPackage, realName, suid, destCodeFile,
26                     destResourceFile, pkg.applicationInfo.nativeLibraryDir,
27                     pkg.applicationInfo.flags, true, false);
28             ......
29         }
30
31         synchronized (mPackages) {
32             ......
33
34             // Add the new setting to mPackages
35             mPackages.put(pkg.applicationInfo.packageName, pkg);
36             ......
37
38             int N = pkg.providers.size();
39             int i;
40             for (i=0; i<N; i++) {
41                 PackageParser.Provider p = pkg.providers.get(i);
42                 ......
43                 mProvidersByComponent.put(new ComponentName(p.info.packageName,
44                     p.info.name), p);
45                 ......
46             }
47
48             N = pkg.services.size();
49             for (i=0; i<N; i++) {
50                 PackageParser.Service s = pkg.services.get(i);
51                 ......
52                 mServices.addService(s);
53                 ......
54             }
55
56             N = pkg.receivers.size();
57             r = null;
```

```
58              for (i=0; i<N; i++) {
59                  PackageParser.Activity a = pkg.receivers.get(i);
60                  ......
61                  mReceivers.addActivity(a, "receiver");
62                  ......
63              }
64
65              N = pkg.activities.size();
66              for (i=0; i<N; i++) {
67                  PackageParser.Activity a = pkg.activities.get(i);
68                  ......
69                  mActivities.addActivity(a, "activity");
70                  ......
71              }
72
73              ......
74          }
75          ......
76
77          return pkg;
78      }
79
80      ......
81  }
```

这个函数主要是完成以下6个工作。

（1）为参数pkg所描述的一个应用程序分配Linux用户ID，如第15行到第27行代码所示。

（2）将参数pkg所指向的一个Package对象保存在PackageManagerService类的成员变量mPackages中，如第35行代码所示。

（3）将参数pkg所描述的一个应用程序的Content Provider组件配置信息保存在PackageManagerService类的成员变量mProvidersByComponent中，如第38行到第46行代码所示。

（4）将参数pkg所描述的一个应用程序的Service组件配置信息保存在PackageManagerService类的成员变量mServices中，如第48行到第54行代码所示。

（5）将参数pkg所描述的一个应用程序的Broadcast Receiver组件配置信息保存在PackageManagerService类的成员变量mReceivers中，如第56行到第63行代码所示。

（6）将参数pkg所描述的一个应用程序的Activity组件配置信息保存在PackageManagerService类的成员变量mActivities中，如第65行到第71行代码所示。

第2个到第6个工作都比较直观，接下来我们主要分析第1个工作的处理过程，即为一个应用程序分配一个Linux用户ID的过程。

第15行的if语句首先检查参数pkg所描述的一个应用程序是否指定了要与其他应用程序共享同一个Linux用户ID。如果指定了，那么接下来第16行和第17行代码就会调用PackageManagerService类的成员变量mSettings所指向的一个Settings对象的成员函数getSharedUserLP来获得这个被共享的Linux用户，并且保存在变量suid中。

Settings类的成员函数getSharedUserLP的实现如下所示。

frameworks/base/core/java/android/content/pm/PackageParser.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     private static final class Settings {
05         ......
06
```

```
07          private final HashMap<String, SharedUserSetting> mSharedUsers =
08                  new HashMap<String, SharedUserSetting>();
09          ......
10
11          SharedUserSetting getSharedUserLP(String name,
12                  int pkgFlags, boolean create) {
13              SharedUserSetting s = mSharedUsers.get(name);
14              if (s == null) {
15                  if (!create) {
16                      return null;
17                  }
18                  s = new SharedUserSetting(name, pkgFlags);
19                  if (MULTIPLE_APPLICATION_UIDS) {
20                      s.userId = newUserIdLP(s);
21                  } else {
22                      s.userId = FIRST_APPLICATION_UID;
23                  }
24                  ......
25                  if (s.userId >= 0) {
26                      mSharedUsers.put(name, s);
27                  }
28              }
29
30              return s;
31          }
32
33          ......
34      }
35
36      ......
37  }
```

参数name用来描述一个共享Linux用户的名称，而参数create用来描述当系统中不存在名称等于name的共享Linux用户时，是否需要新创建一个。

在前面的Step 6中提到，系统中的所有共享Linux用户都是使用一个SharedUserSetting对象来描述的，并且以它们的名称为关键字保存在Settings类的成员变量mSharedUsers中。因此，第13行代码首先以参数name为关键字在里面查找是否存在一个对应的SharedUserSetting对象。如果存在，即第14行的if语句为false，那么第30行代码就将这个SharedUserSetting对象返回给调用者；否则，接下来第15行到第27行代码就会检查是否需要使用参数name来创建一个对应的SharedUserSetting对象，以便可以返回给调用者。

第15行的if语句首先检查参数create的值是否等于true。如果不等于true，那么接下来第16行代码就会直接返回一个null值给调用者，表示找不到名称等于name的共享Linux用户；否则，第18行到第27行代码就会首先使用参数name来创建一个SharedUserSetting对象，即创建一个名称等于name的共享Linux用户，然后调用Settings类的成员函数newUserIdLP来为这个SharedUserSetting对象分配一个Linux用户ID，最后再将这个SharedUserSetting对象保存在Settings类的成员变量mSharedUsers中。在接下来的Step 13中，我们再分析Settings类的成员函数newUserIdLP的实现。

注意

MULTIPLE_APPLICATION_UIDS是PackageManagerService类的一个静态成员变量，它的值被设置为true，表示要为系统中的每一个应用程序分配一个不同的Linux用户ID，但是不包括那些要求与其他应用程序共享同一个Linux用户ID的应用程序。如果它的值等于false，那么就表示系统中的所有应用程序都共用一个值为FIRST_APPLICATION_UID 的Linux用户ID。

回到PackageManagerService类的成员函数scanPackageLI中，接下来第25行到第27行代码就会继续调用Settings类的成员函数getPackageLP来为参数pkg所描述的一个应用程序分配一个Linux用户ID。

Step 12：Settings. getPackageLP

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     private static final class Settings {
05         ......
06
07         PackageSetting getPackageLP(PackageParser.Package pkg, PackageSetting origPackage,
08                 String realName, SharedUserSetting sharedUser, File codePath, File resourcePath,
09                 String nativeLibraryPathString, int pkgFlags, boolean create, boolean add) {
10             final String name = pkg.packageName;
11             PackageSetting p = getPackageLP(name, origPackage, realName, sharedUser, codePath,
12                     resourcePath, nativeLibraryPathString, pkg.mVersionCode, pkgFlags, create, add);
13             return p;
14         }
15
16         ......
17     }
18
19     ......
20 }
```

第10行代码首先将Package对象pkg的成员变量packageName的值保存在变量name中，接着第11行和第12行代码再调用Settings类的另外一个重载版本的成员函数getPackageLP来为Package名称等于name的应用程序分配一个Linux用户ID。

Settings类的另外一个重载版本的成员函数getPackageLP的实现如下所示。

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     private static final class Settings {
05         ......
06
07         private final HashMap<String, PackageSetting> mPackages =
08                 new HashMap<String, PackageSetting>();
09         ......
10
11         private PackageSetting getPackageLP(String name, PackageSetting origPackage,
12                 String realName, SharedUserSetting sharedUser, File codePath, File resourcePath,
13                 String nativeLibraryPathString, int vc, int pkgFlags, boolean create, boolean add) {
14             PackageSetting p = mPackages.get(name);
15             if (p != null) {
16                 ......
17                 if (p.sharedUser != sharedUser) {
18                     ......
19                     p = null;
20                 }
21                 ......
22             }
23             if (p == null) {
24                 // Create a new PackageSettings entry. this can end up here because
25                 // of code path mismatch or user id mismatch of an updated system partition
26                 if (!create) {
```

```
27                      return null;
28                  }
29                  if (origPackage != null) {
30                      // We are consuming the data from an existing package.
31                      p = new PackageSetting(origPackage.name, name, codePath, resourcePath,
32                              nativeLibraryPathString, vc, pkgFlags);
33                      ......
34                      p.copyFrom(origPackage);
35                      ......
36                      p.sharedUser = origPackage.sharedUser;
37                      p.userId = origPackage.userId;
38                      p.origPackage = origPackage;
39                      ......
40                  } else {
41                      p = new PackageSetting(name, realName, codePath, resourcePath,
42                              nativeLibraryPathString, vc, pkgFlags);
43                      ......
44                      p.sharedUser = sharedUser;
45                      if (sharedUser != null) {
46                          p.userId = sharedUser.userId;
47                      } else if (MULTIPLE_APPLICATION_UIDS) {
48                          // Clone the setting here for disabled system packages
49                          PackageSetting dis = mDisabledSysPackages.get(name);
50                          if (dis != null) {
51                              ......
52                              p.userId = dis.userId;
53                              ......
54                          } else {
55                              // Assign new user id
56                              p.userId = newUserIdLP(p);
57                          }
58                      } else {
59                          p.userId = FIRST_APPLICATION_UID;
60                      }
61                  }
62                  ......
63
64                  if (add) {
65                      // Finish adding new package by adding it and updating shared
66                      // user preferences
67                      addPackageSettingLP(p, name, sharedUser);
68                  }
69              }
70              return p;
71          }
72
73          ......
74      }
75
76      ......
77  }
```

在前面的Step 4中提到，系统中的所有应用程序的安装信息都使用一个PackageSetting对象来描述，并且保存在Settings类的成员变量mPackages所描述的一个HashMap中。

注意

这些安装信息有可能是来自于全新安装的应用程序的，还有可能是来自于上一次安装的应用程序的。

参数name用来描述一个应用程序的Package名称，第14行代码以它为关键字在Settings类的成员变量mPackages中查找是否存在一个对应的PackageSetting对象p。如果存在，那么接下来第17行的if语句就会检查这个PackageSetting对象p是否是用来描述一个与其他应用程序共享同一个Linux用户ID的应用程序的安装信息的。如果是，那么第17行的if语句还需要继续检查这个PackageSetting对象p的成员变量sharedUser所描述的一个共享Linux用户是否与参数sharedUser所描述的一个SharedUserSetting对象相同。如果不相同，即第17行的if语句为true，那么这个PackageSetting对象p就不能用来描述Package名称等于name的应用程序的安装信息。因此，第19行代码会将它的值重新设置为null，以便接下来可以为Package名称等于name的应用程序创建一个PackageSetting对象。

第23行代码检查前面所获得的一个PackageSetting对象p的值是否等于null。如果不等于null，那么接下来第70行代码就会直接将它返回给调用者；否则，第26行到第68行代码就会检查是否需要为Package名称等于name的应用程序创建一个PackageSetting对象，以便可以返回给调用者。

第26行的if语句检查参数create的值是否等于true。如果不等于true，那么就说明不需要为Package名称等于name的应用程序创建一个PackageSetting对象。因此，第27行代码就会直接返回一个null给调用者；否则，第29行到第68行代码就会为Package名称等于name的应用程序创建一个PackageSetting对象。

第29行的if语句检查参数origPackage的值是否不等于null。如果不等于null，那么就说明Package名称等于name的应用程序在系统中有一个旧的版本。在这种情况下，接下来第31行到第38行代码就会为这个旧版本的应用程序的Package名称以及Linux用户ID创建一个新的PackageSetting对象p。

如果参数origPackage的值等于null，即第29行的if语句为false，那么就说明Package名称等于name的应用程序是一个全新安装的应用程序。在这种情况下，接下来第41行到第60行代码就会全部使用函数的参数来为Package名称等于name的应用程序创建一个新的PackageSetting对象p。

第45行的if语句检查Package名称等于name的应用程序是否指定了要与其他的应用程序共享同一个Linux用户ID。如果指定了，那么参数sharedUser就指向了被共享的Linux用户。因此，第46行代码就会将它的Linux用户ID设置为前面所创建的PackageSetting对象p的成员变量userId的值，以便可以用来描述Package名称等于name的应用程序的Linux用户ID。

如果Package名称等于name的应用程序没有指定要与其他的应用程序共享同一个Linux用户ID，即第45行的if语句为false，那么第47行的if语句就会检查PackageManagerService类的静态成员变量MULTIPLE_APPLICATION_UIDS的值是否等于true。如果等于true，那么接下来第49行到第57行代码就会为Package名称等于name的应用程序分配一个新的Linux用户ID；否则，第59行代码就会简单地将Package名称等于name的应用程序的Linux用户ID设置为FIRST_APPLICATION_UID。

第50行的if语句检查Package名称等于name的应用程序是否是一个禁用的系统应用程序。如果是，那么就不需要为Package名称等于name的应用程序分配一个新的Linux用户ID了，直接使用它原来的Linux用户ID即可，如第52行代码所示；否则，第56行代码就会调用Settings类的成员函数newUserIdLP来分配一个新的Linux用户ID，并且保存在新创建的PackageSetting对象p的成员变量userId中，以便可以用来描述Package名称等于name的应用程序的Linux用户ID。

第64行的if语句检查参数add的值是否等于true。如果等于true，那么第67行代码就会调用Settings类的成员函数addPackageSettingLP将新创建的PackageSetting对象p保存在Settings类的成员变量mPackages中，以便可以表示它描述的应用程序已经成功地安装在系统中了。

接下来，我们继续分析Settings类的成员函数newUserIdLP的实现，以便可以了解它是如何为一个新安装的应用程序分配一个Linux用户ID的。

Step 13：Settings.newUserIdLP

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     private static final class Settings {
05         ......
06
07         private final ArrayList<Object> mUserIds = new ArrayList<Object>();
08         ......
09
10         private int newUserIdLP(Object obj) {
11             // Let's be stupidly inefficient for now...
12             final int N = mUserIds.size();
13             for (int i=0; i<N; i++) {
14                 if (mUserIds.get(i) == null) {
15                     mUserIds.set(i, obj);
16                     return FIRST_APPLICATION_UID + i;
17                 }
18             }
19
20             // None left?
21             if (N >= MAX_APPLICATION_UIDS) {
22                 return -1;
23             }
24
25             mUserIds.add(obj);
26             return FIRST_APPLICATION_UID + N;
27         }
28
29         ......
30     }
31
32     ......
33 }
```

在前面的Step 4中提到，Settings类的成员变量mUserIds用来保存系统中所有已经分配的Linux用户ID，它是一个类型为ArrayList的列表。如果保存在这个列表中的第i个位置的一个Object对象的值等于null，那么就说明值等于（FIRST_APPLICATION_UID + i）的Linux用户ID尚未分配出去，因此，我们就可以将它分配给新安装的应用程序使用，如第13行到第18行的for循环所示。

如果不能在Settings类的成员变量mUserIds中找到一个空闲的Linux用户ID，并且系统已经分配出去的Linux用户ID的数量还没有达到最大值（MAX_APPLICATION_UIDS），即第21行的if语句为false，那么接下来第25行和第26行代码就会将值等于（FIRST_APPLICATION_UID + N）的Linux用户ID分配给新安装的应用程序使用，其中，变量N表示之前已经分配出去的Linux用户ID的数量。

这一步执行完成之后，一个应用程序就成功地安装到系统中了。返回到前面的Step 1中，即PackageManagerService类的构造函数中，当系统中的所有应用程序都安装完成之后，接下来第14步到第17步就会调用PackageManagerService类的成员函数updatePermissionsLP来为前面所安装的应用程序分配Linux用户组ID，即授予它们所申请的资源访问权限，以及调用Settings类的成员函数writeLP将这些应用程序的安装信息保存在本地文件中。这个过程如图16–3所示。

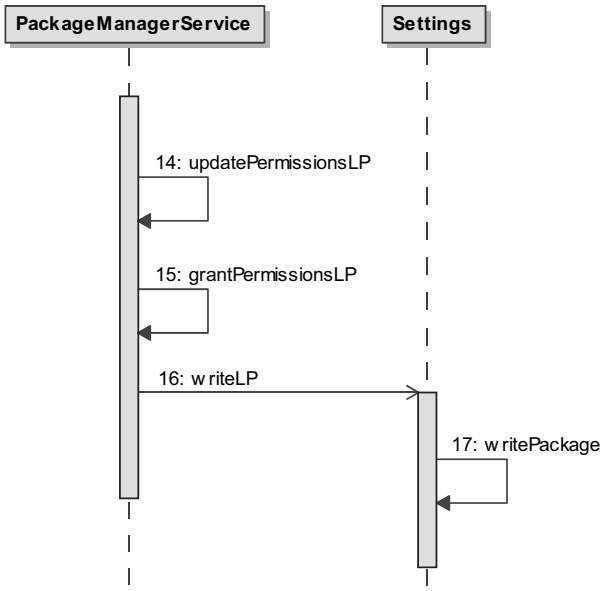


图16–3　应用程序的安装过程——Step 14至Step 17

Step 14：PackageManagerService. updatePermissionsLP

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     final HashMap<String, PackageParser.Package> mPackages =
05             new HashMap<String, PackageParser.Package>();
06     ......
07
08     private void updatePermissionsLP(String changingPkg,
09             PackageParser.Package pkgInfo, boolean grantPermissions,
10             boolean replace, boolean replaceAll) {
11         ......
12
13         if (grantPermissions) {
14             for (PackageParser.Package pkg : mPackages.values()) {
15                 if (pkg != pkgInfo) {
16                     grantPermissionsLP(pkg, replaceAll);
17                 }
18             }
19         }
20
21         ......
22     }
23
24     ......
25 }
```

所有已经安装了的应用程序都保存在PackageManagerService类的成员变量mPackages中，当参数grantPermissions的值等于true时，第14行到第18行的for循环就会依次调用PackageManagerService类的成员函数grantPermissionsLP来为它们分配Linux用户组ID，以便它们可以获得所申请的资源访问权限。

Step 15：PackageManagerService. grantPermissionsLP

在分析这个函数的实现之前，我们首先了解配置文件AndroidManifest.xml中的uses–permission标签

和Android应用程序的Linux用户组ID的关系。

假设一个应用程序需要使用照相设备，那么它就需要在它的配置文件AndroidManifest.xml中添加下面这一行信息：

```
<uses-permission android:name="android.permission.CAMERA" />
```

从前面的Step 9可以知道，Package管理服务PackageManagerService在解析这个配置文件时，会将这个uses-permission标签中的android:name属性的值“android.permission.CAMERA”取出来，并且保存在一个对应的Package对象的成员变量requestedPermissions所描述的一个资源访问权限列表中。

Package管理服务PackageManagerService在启动时，会调用PackageManagerService类的成员函数readPermissions来解析保存在设备上的/system/etc/permissions/platform.xml文件中的内容。这个文件的内容是以xml格式保存的，里面包含了一系列的permission标签，用来描述系统中的资源访问权限列表，它们的格式如下所示：

```
<permission name="android.permission.CAMERA" >
    <group gid="camera" />
</permission>
```

这个permission标签表示使用名称为“camera”的Linux用户组来描述名称为“android.permission.CAMERA”的资源访问权限。知道了一个Linux用户组的名称之后，我们就可以调用由Linux内核提供的函数getgrnam来获得对应的Linux用户组ID，这样就可以将一个资源访问权限与一个Linux用户组ID关联起来。

Package管理服务PackageManagerService会为这个文件中的每一个permission标签创建一个BasePermission对象，并且以这个标签中的name属性值作为关键字，将这些BasePermission对象保存在PackageManagerService类的成员变量mSettings所指向的一个Settings对象的成员变量mPermissions所描述的一个HashMap中。

由于一个permission标签可以包含多个group子标签，即一个资源访问权限名称可以用来描述多个Linux用户组，因此，每一个BasePermission对象内部都有一个gids数组，用来保存所有与它对应的Linux用户组ID。

在设备上的/system/etc/permissions/platform.xml文件中，还包含了一些全局的group标签。这些全局的group标签与permission标签是同一级的，用来描述系统中所有的应用程序默认都具有的资源访问权限。Package管理服务PackageManagerService会将这些全局的group标签所描述的Linux用户组ID保存在PackageManagerService类的成员变量mGlobalGids所描述的一个数组中。

了解了这些背景知识之后，接下来我们就分段来阅读PackageManagerService类的成员函数grantPermissionsLP的代码，如下所示。

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
001 class PackageManagerService extends IPackageManager.Stub {
002     ......
003
004     private void grantPermissionsLP(PackageParser.Package pkg, boolean replace) {
005         final PackageSetting ps = (PackageSetting)pkg.mExtras;
006         ......
007
008         final GrantedPermissions gp = ps.sharedUser != null ? ps.sharedUser : ps;
009         ......
010
011         if (replace) {
012             ......
013             if (gp == ps) {
```

```
014                 gp.grantedPermissions.clear();
015                 gp.gids = mGlobalGids;
016             }
017         }
018
019         if (gp.gids == null) {
020             gp.gids = mGlobalGids;
021         }
```

参数pkg所描述的一个应用程序的安装信息保存在它的成员变量mExtras中，第5行代码首先将它取出来，并且保存在PackageSetting对象ps中，以便接下来可以根据这些安装信息来为这个应用程序分配Linux用户组ID。

第8行代码检查参数pkg所描述的一个应用程序是否指定了要与其他应用程序共享同一个Linux用户ID。如果指定了，那么前面所获得的PackageSetting对象ps的成员变量sharedUser就用来描述被共享的Linux用户。在这种情况下，参数pkg所描述的一个应用程序所获得的资源访问权限就与它所共享的Linux用户所具有的资源访问权限相同。第8行代码执行完成之后，参数pkg所描述的一个应用程序所申请的资源访问权限就使用GrantedPermissions对象gp来描述。

第11行的if语句检查参数replace的值是否等于true。如果等于true，那么第13行的if语句就会继续检查参数pkg所描述的一个应用程序是否指定了要与其他应用程序共享同一个Linux用户ID。如果没有指定，即第13行的if语句为true，那么接下第14行和第15行代码就会将已经授予给这个应用程序的资源访问权限替换为系统中所有应用程序默认都具有的资源访问权限。

注意

一个应用程序已经被授予的资源访问权限保存在与它所对应的一个GrantedPermissions对象的成员变量gids所描述的一个数组中。

第19行的if语句检查GrantedPermissions对象gp的成员变量gids的值是否等于null。如果等于null，那么就说明系统尚未为参数pkg所描述的一个应用程序分配过Linux用户组ID。在这种情况下，接下来第20行代码就会将保存在PackageManagerService类的成员变量mGlobalGids中的Linux用户组ID复制到GrantedPermissions对象gp的成员变量gids中，以便可以给参数pkg所描述的一个应用程序授予默认的资源访问权限。

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
022         final int N = pkg.requestedPermissions.size();
023         for (int i=0; i<N; i++) {
024             String name = pkg.requestedPermissions.get(i);
025             BasePermission bp = mSettings.mPermissions.get(name);
026             ......
027             if (bp != null && bp.packageSetting != null) {
028                 final String perm = bp.name;
029                 boolean allowed;
030                 boolean allowedSig = false;
031                 if (bp.protectionLevel == PermissionInfo.PROTECTION_NORMAL
032                         || bp.protectionLevel == PermissionInfo.PROTECTION_DANGEROUS) {
033                     allowed = true;
034                 } else if (bp.packageSetting == null) {
035                     // This permission is invalid; skip it.
036                     allowed = false;
037                 } else if (bp.protectionLevel == PermissionInfo.PROTECTION_SIGNATURE
038                         || bp.protectionLevel == PermissionInfo.PROTECTION_SIGNATURE_OR_SYSTEM) {
039                     allowed = (checkSignaturesLP(
040                             bp.packageSetting.signatures.mSignatures, pkg.mSignatures)
```

```
041                                     == PackageManager.SIGNATURE_MATCH)
042                             || (checkSignaturesLP(mPlatformPackage.mSignatures, pkg.mSignatures)
043                                     == PackageManager.SIGNATURE_MATCH);
044                     if (!allowed && bp.protectionLevel
045                             == PermissionInfo.PROTECTION_SIGNATURE_OR_SYSTEM) {
046                         if (isSystemApp(pkg)) {
047                             // For updated system applications, the signatureOrSystem permission
048                             // is granted only if it had been defined by the original application.
049                             if (isUpdatedSystemApp(pkg)) {
050                                 PackageSetting sysPs = mSettings.getDisabledSystemPkg(
051                                         pkg.packageName);
052                                 final GrantedPermissions origGp = sysPs.sharedUser != null
053                                         ? sysPs.sharedUser : sysPs;
054                                 if (origGp.grantedPermissions.contains(perm)) {
055                                     allowed = true;
056                                 } else {
057                                     allowed = false;
058                                 }
059                             } else {
060                                 allowed = true;
061                             }
062                         }
063                     }
064                     if (allowed) {
065                         allowedSig = true;
066                     }
067                 } else {
068                     allowed = false;
069                 }
```

第23行到后面第113行的for循环依次检查参数pkg所描述的一个应用程序所申请的每一个资源访问权限是否合法。如果合法，那么就给它分配相应的Linux用户组ID。

第24行代码首先获得当前正在检查的资源访问权限的名称，接着第25行代码根据这个名称在PackageManagerService类的成员变量mSettings所指向的一个Settings对象的成员变量mPermissions所描述的一个HashMap中检查是否存在一个相应的BasePermission对象bp。如果存在，那么第28行到第66行代码就会进一步判断当前正在检查的资源访问权限是否合法；否则，第68行代码就会直接将变量allowed的值设置为false，以便可以表示当前正在检查的资源访问权限是不合法的。

如果BasePermission对象bp所描述的一个资源访问权限的保护级别为PROTECTION_NORMAL或者PROTECTION_DANGEROUS，即第31行和第32行的if语句为true，那么就表示当前正在检查的资源访问权限是合法的，这时候就会将变量allowed的值设置为true，如第33行代码所示。

如果BasePermission对象bp所描述的一个资源访问权限的保护级别为PROTECTION_SIGNATURE或者PROTECTION_SIGNATURE_OR_SYSTEM，即第37行和第38行的if语句为true，那么就说明需要结合参数pkg所描述的一个应用程序的签名来判断当前正在检查的资源访问权限是否合法，如第39行到第66行代码所示。

第39行到第43行代码首先调用PackageManagerService类的成员函数checkSignaturesLP来检查参数pkg所描述的一个应用程序的签名是否与BasePermission对象bp所要求的签名或者平台Package所要求的签名匹配。如果匹配，那么变量allowed的值就会被设置为true；否则，第44行到第63行代码还需要进一步判断当前正在检查的资源访问权限是否合法。

注意

平台Package是一个特殊的应用程序包，它的Package名称等于“android”，保存在PackageManagerService类的成员变量mPlatformPackage中。

如果前面得到的变量allowed的值等于false，并且BasePermission对象bp所描述的一个资源访问权限的保护级别为PROTECTION_SIGNATURE_OR_SYSTEM，即第44行和第45行的if语句为true，那么第46行的if语句就会继续检查参数pkg所描述的一个应用程序是否是一个系统应用程序。如果是，那么第49行的if语句又会检查这个应用程序是否是用来更新一个旧版本的系统应用程序的。如果不是，那么第60行代码就会直接将变量allowed的值修改为true；否则，第50行到第58行代码就需要继续检查这个旧版本的系统应用程序是否具有当前正在检查的资源访问权限。如果具有，那么第55行代码也会将变量allowed的值修改为true。

函数执行到第64行的if语句时，如果变量allowed的值等于true，那么就说明参数pkg所描述的一个应用程序的签名与要求的签名是匹配的，因此，第65行代码就会将变量allowedSig的值也设置为true。

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
070                 if (allowed) {
071                     if ((ps.pkgFlags&ApplicationInfo.FLAG_SYSTEM) == 0
072                             && ps.permissionsFixed) {
073                         // If this is an existing, non-system package, then
074                         // we can't add any new permissions to it.
075                         if (!allowedSig && !gp.grantedPermissions.contains(perm)) {
076                             allowed = false;
077                             // Except...  if this is a permission that was added
078                             // to the platform (note: need to only do this when
079                             // updating the platform).
080                             final int NP = PackageParser.NEW_PERMISSIONS.length;
081                             for (int ip=0; ip<NP; ip++) {
082                                 final PackageParser.NewPermissionInfo npi
083                                         = PackageParser.NEW_PERMISSIONS[ip];
084                                 if (npi.name.equals(perm)
085                                         && pkg.applicationInfo.targetSdkVersion < npi.sdkVersion) {
086                                     allowed = true;
087                                     ......
088                                     break;
089                                 }
090                             }
091                         }
092                     }
093                     if (allowed) {
094                         if (!gp.grantedPermissions.contains(perm)) {
095                             ......
096                             gp.grantedPermissions.add(perm);
097                             gp.gids = appendInts(gp.gids, bp.gids);
098                         } else if (!ps.haveGids) {
099                             gp.gids = appendInts(gp.gids, bp.gids);
100                         }
101                     }
102                     ......
103                 } else {
104                     if (gp.grantedPermissions.remove(perm)) {
105                         ......
106                         gp.gids = removeInts(gp.gids, bp.gids);
107                         ......
108                     }
109                     ......
110                 }
111             }
112             ......
113         }
114
115         ......
116     }
117
118     ......
119 }
```

函数执行到第70行的if语句时，如果变量allowed的值等于false，那么就可以直接说明正在检查的资源访问权限是不合法的。因此，接下来第104行到第108行代码就会将它从参数pkg所描述的一个应用程序的资源访问权限列表中移除；否则，第71行到第101行代码还需要继续判断正在检查的资源访问权限是否合法。

第71行和第72行的if语句如果发现参数pkg所描述的不是一个系统应用程序，并且它是一个在系统中已经存在的应用程序，那么第75行到第91行代码就会根据参数pkg所描述的一个应用程序的签名是否与要求的签名匹配，以及原来已经存在的应用程序是否具有当前正在检查的资源访问权限，来决定当前正在检查的资源访问权限是否合法。如果签名匹配，即变量allowedSig的值等于true，或者原来已经存在的应用程序是具有当前正在检查的资源访问权限的，那么当前正在检查的资源访问权限就是合法的。如果签名不匹配，并且原来已经存在的应用程序不具有当前正在检查的资源访问权限，即当前正在检查的资源访问权限是一个新增的资源访问权限，那么第76行代码就会将变量allowed的值修改为false，以便可以表示当前正在检查的资源访问权限是不合法的。不过，在这种情况下，有一个例外。如果当前正在检查的资源访问权限是一个由于系统升级而新增的资源访问权限，那么也会认为当前正在检查的资源访问权限是合法的，这样就可以保证应用程序的兼容性，如第80行到第90行代码所示。

注意

系统由于升级而新增的资源访问权限保存在PackageParser类的静态成员变量NEW_PERMISSIONS所描述的一个NewPermissionInfo数组中。

函数执行到第93行的if语句时，如果变量allowed的值等于true，那么就说明当前正在检查的资源访问权限是合法的。因此，接下来第94行到第100行代码就可以将它授予给参数pkg所描述的一个应用程序，即可以为参数pkg所描述的一个应用程序分配一个相应的Linux用户组ID。

这一步执行完成之后，回到前面的Step 1中，即PackageManagerService类的构造函数中，这时候Package管理服务PackageManagerService就为前面所安装的应用程序都分配好Linux用户组ID了。接下来它就会继续调用Settings类的成员函数writeLP将这些应用程序的安装信息保存在本地文件中。

Step 16：Settings.writeLP

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     private static final class Settings {
05         ......
06
07         private final HashMap<String, PackageSetting> mPackages =
08                 new HashMap<String, PackageSetting>();
09         ......
10
11         private final HashMap<String, SharedUserSetting> mSharedUsers =
12                 new HashMap<String, SharedUserSetting>();
13         ......
14
15         void writeLP() {
16             ......
17
18             if (mSettingsFilename.exists()) {
19                 ......
20
21                 if (!mBackupSettingsFilename.exists()) {
```

```
22                  if (!mSettingsFilename.renameTo(mBackupSettingsFilename)) {
23                      ......
24                      return;
25                  }
26              } else {
27                  mSettingsFilename.delete();
28                  ......
29              }
30          }
31          ......
32
33          try {
34              FileOutputStream fstr = new FileOutputStream(mSettingsFilename);
35              BufferedOutputStream str = new BufferedOutputStream(fstr);
36
37              //XmlSerializer serializer = XmlUtils.serializerInstance();
38              XmlSerializer serializer = new FastXmlSerializer();
39              serializer.setOutput(str, "utf-8");
40              serializer.startDocument(null, true);
41              serializer.setFeature("http://xmlpull.org/v1/doc/features.html#indent-output", true);
42
43              serializer.startTag(null, "packages");
44              ......
45
46              for (PackageSetting pkg : mPackages.values()) {
47                  writePackage(serializer, pkg);
48              }
49              ......
50
51              for (SharedUserSetting usr : mSharedUsers.values()) {
52                  serializer.startTag(null, "shared-user");
53                  serializer.attribute(null, "name", usr.name);
54                  serializer.attribute(null, "userId",
55                          Integer.toString(usr.userId));
56                  usr.signatures.writeXml(serializer, "sigs", mPastSignatures);
57                  serializer.startTag(null, "perms");
58                  for (String name : usr.grantedPermissions) {
59                      serializer.startTag(null, "item");
60                      serializer.attribute(null, "name", name);
61                      serializer.endTag(null, "item");
62                  }
63                  serializer.endTag(null, "perms");
64                  serializer.endTag(null, "shared-user");
65              }
66              ......
67
68              serializer.endTag(null, "packages");
69
70              serializer.endDocument();
71
72              str.flush();
73              FileUtils.sync(fstr);
74              str.close();
75
76              // New settings successfully written, old ones are no longer
77              // needed.
78              mBackupSettingsFilename.delete();
79              ......
80
81          } catch(XmlPullParserException e) {
82              ......
```

```
83              } catch(java.io.IOException e) {
84                  ......
85              }
86
87              ......
88          }
89
90          ......
91      }
92
93      ......
94  }
```

Package管理服务PackageManagerService要保存的应用程序安装信息比较多，不过我们只关注与应用程序的Linux用户ID相关的安装信息的保存过程。

从前面的Step 2中可以知道，应用程序安装信息是保存在Settings类的成员变量mSettingsFilename和mBackupSettingsFilename所描述的两个文件中的，其中，后者是前者的备份文件，它们对应的系统路径分别为/data/system/packages.xml和/data/system/packages-backup.xml。

第18行的if语句首先判断文件/data/system/packages.xml是否存在。如果存在，那么第21行的if语句再判断备份文件/data/system/packages-backup.xml是否也存在。如果不存在，那么第22行代码就会将文件/data/system/packages.xml重命名为/data/system/packages-backup.xml；否则，第27行代码就会将/data/system/packages.xml删除。这段代码的目的就是将保存在文件/data/system/packages.xml中的内容备份到文件/data/system/packages-backup.xml中，这样就不用担心将新的应用程序安装信息写入到文件/data/system/packages.xml中时出现意外，因为在这种情况下，Package管理服务PackageManagerService可以通过文件/data/system/packages-backup.xml来恢复上一次的应用程序安装信息。

第34行到第43行代码首先通过XmlSerializer对象serializer来初始化文件/data/system/packages.xml的内容，以便后面可以写入新的应用程序安装信息。

第46行到第48行的for循环依次调用Settings类的成员函数writePackage将保存在Settings类的成员变量mPackages中的每一个应用程序的安装信息写入到文件/data/system/packages.xml中。在接下来的Step 17中，我们再分析Settings类的成员函数writePackage的实现。

第51行到第65行的for循环依次将保存在Settings类的成员变量mSharedUsers中的每一个共享Linux用户的信息写入到/data/system/packages.xml文件中，其中，每一个共享Linux用户的信息都是使用一个shared-user标签来描述的，这个标签包含的信息有共享Linux用户的名称、ID及其所请求的资源访问权限等。

Package管理服务PackageManagerService成功地将新的应用程序安装信息写入到/data/system/packages.xml文件之后，备份文件/data/system/packages-backup.xml就不需要了，因此，最后第78行代码就将它删除。

接下来，我们继续分析Settings类的成员函数writePackage的实现，以便可以了解它是如何将分配给应用程序的Linux用户ID保存在文件/data/system/packages.xml中的。

Step 17：Settings. writePackage

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
04     private static final class Settings {
05         ......
06
07         void writePackage(XmlSerializer serializer, final PackageSetting pkg)
08                 throws java.io.IOException {
```

```
09              serializer.startTag(null, "package");
10              serializer.attribute(null, "name", pkg.name);
11              ......
12
13              if (pkg.sharedUser == null) {
14                  serializer.attribute(null, "userId",
15                          Integer.toString(pkg.userId));
16              } else {
17                  serializer.attribute(null, "sharedUserId",
18                          Integer.toString(pkg.userId));
19              }
20              ......
21
22              serializer.endTag(null, "package");
23          }
24
25          ......
26      }
27
28      ......
29  }
```

每一个应用程序的安装信息都是使用一个package标签来描述的，其中，这个标签的属性name用来描述一个应用程序的Package名称；而属性userId和sharedUserId用来描述分配给一个应用程序的Linux用户ID和共享Linux用户ID。

注意

属性userId和sharedUserId是互斥的，即它们之间只有一个会存在于一个package标签中，因为一个应用程序要么单独地使用一个Linux用户ID，要么与其他应用程序共享同一个Linux用户ID。

这一步执行完成之后，Package管理服务PackageManagerService就将每一个应用程序所使用的Linux用户ID保存起来了，这样以后无论如何重新安装一个应用程序，Package管理服务PackageManagerService都可以保证这个应用程序所使用的Linux用户ID是不变的。

在分析应用程序的安装过程中，我们着重描述了应用程序的Linux用户ID和Linux用户组ID是如何分配的，这是因为Android系统正好是通过Linux用户ID和Linux用户组ID来防止一个应用程序破坏其他应用程序的数据，以及非法访问系统的特权资源的。

这是如何做到的呢？从前面第12章的内容可以知道，系统中的应用程序进程是由Activity管理服务ActivityManagerService请求Zygote进程来创建的，如下所示。

frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02          implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03      ......
04
05      private final void startProcessLocked(ProcessRecord app,
06                  String hostingType, String hostingNameStr) {
07          ......
08
09          try {
10              int uid = app.info.uid;
11              int[] gids = null;
12              try {
13                  gids = mContext.getPackageManager().getPackageGids(
14                      app.info.packageName);
```

```
15              } catch (PackageManager.NameNotFoundException e) {
16                  ......
17              }
18              ......
19
20              int pid = Process.start("android.app.ActivityThread",
21                  mSimpleProcessManagement ? app.processName : null, uid, uid,
22                  gids, debugFlags, null);
23
24              ......
25      }
26
27      ......
28  }
```

第10行到第17行代码首先获得要创建的应用程序进程的Linux用户ID和Linux用户组ID，接着第20行到第23行代码再调用Process类的静态成员函数start来请求Zygote进程创建一个应用程序进程。在调用的过程中，会通过第3个、第4个和第5个参数来指定新创建的应用程序进程的Linux用户ID和Linux用户组ID。这样就可以将一个应用程序运行在一个受限的进程中，防止它破坏其他应用程序的数据，以及非法访问系统的特权资源。

至此，应用程序在系统启动时的安装过程就分析完成了。如果一个应用程序是在系统启动完成之后再安装的，那么安装过程也是类似的，只不过最开始的安装入口不同而已。例如，当我们从网络上下载一个应用程序到设备上安装时，系统会调用PackageManager类的成员函数installPackage来启动这个应用程序的安装过程，而PackageManager类的成员函数installPackage最终是通过PackageManagerService类的成员函数scanPackageLI来完成这个应用程序的安装过程的。

接下来，我们继续分析安装在系统中的应用程序是如何显示在屏幕上的。

16.2 应用程序的显示过程

Android系统提供了一个默认的Home应用程序——Launcher，用来显示系统中已经安装了的应用程序，它是由System进程负责启动的，同时它也是系统中第一个启动的应用程序。

将系统中已经安装了的应用程序显示出来的目的是为了让用户有一个统一的入口来启动它们。在前面的7.2小节中提到，一个根Acitivity组件的启动过程就代表了一个应用程序的启动过程。因此，应用程序Launcher在显示一个应用程序时，只需要获得它的根Activity组件信息即可。

一个应用程序的根Activity组件的类型被定义为CATEGORY_LAUNCHER，同时它的Action名称被定义为ACTION_MAIN，这样应用程序Launcher就可以通过这两个条件来请求Package管理服务PackageManagerService返回系统中所有已经安装了的应用程序的根Activity组件信息。

获得了系统中已经安装了的应用程序的根Activity组件信息之后，应用程序Launcher就会分别将它们封装成一个快捷图标，并且显示在系统的屏幕中，这样用户就可以通过点击这些快捷图标来启动相应的应用程序了。

接下来，我们就通过分析应用程序Launcher的启动过程来说明系统中已经安装了的应用程序的显示过程。

从前面第11章的内容可以知道，System进程是由Zygote进程负责启动的，而System进程在启动过程中，又会创建一个ServerThread线程来启动系统中的关键服务。当系统中的关键服务都启动起来之后，这个ServerThread线程就会通知Activity管理服务ActivityManagerService将应用程序Launcher启动起

来。这个启动过程一共分为13个步骤，下面我们就详细分析每一个步骤。

第1步到第5步如图16–4所示。

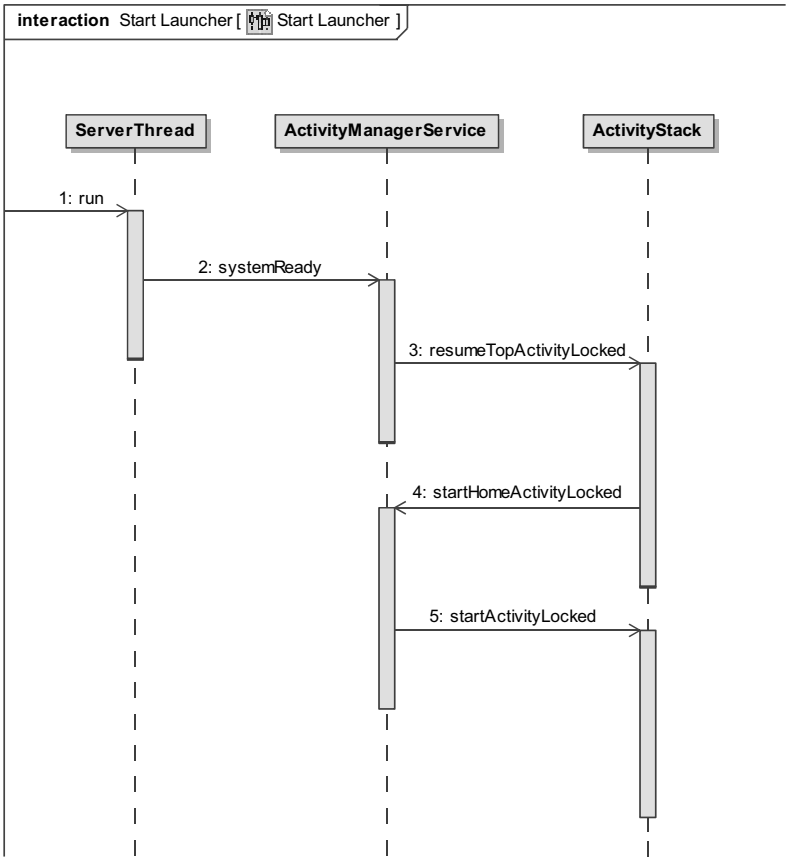


图16–4　应用程序的显示过程——Step 1至Step 5

Step 1：ServerThread.run

frameworks/base/services/java/com/android/server/SystemServer.java

```
01 class ServerThread extends Thread {
02     ......
03
04     public void run() {
05         ......
06
07         Looper.prepare();
08         ......
09
10         try {
11             ......
12
13             context = ActivityManagerService.main(factoryTest);
14             ......
15
16             pm = PackageManagerService.main(context,
17                     factoryTest != SystemServer.FACTORY_TEST_OFF);
18             ......
19
20         } catch (RuntimeException e) {
21             ......
22         }
23
```

```
24          ((ActivityManagerService)ActivityManagerNative.getDefault())
25                  .systemReady(new Runnable() {
26              public void run() {
27                  Slog.i(TAG, "Making services ready");
28                  ......
29              }
30          });
31          ......
32
33          Looper.loop();
34          ......
35      }
36  }
```

第11行到第19行代码用来启动系统中的关键服务。其中，第13行和第16行代码分别用来启动Activity管理服务ActivityManagerService和Package管理服务PackageManagerService。

从前面16.1小节的内容可以知道，当Package管理服务PackageManagerService启动起来之后，系统中的应用程序就安装完成了。因此，接下来第24行到第30行代码就可以通知前面已经启动起来的Activity管理服务ActivityManagerService将应用程序Launcher启动起来，这是通过调用ActivityManagerService类的成员函数systemReady来实现的。

Step 2：ActivityManagerService.systemReady

frameworks/base/services/java/com/android/server/am/ActivityManagerServcie.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02          implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03      ......
04
05      public ActivityStack mMainStack;
06      ......
07
08      public void systemReady(final Runnable goingCallback) {
09          ......
10
11          synchronized (this) {
12              ......
13
14              mMainStack.resumeTopActivityLocked(null);
15          }
16      }
17
18      ......
19  }
```

ActivityManagerService类的成员变量mMainStack指向了一个ActivityStack对象，这个ActivityStack对象是用来描述系统的Activity组件堆栈的，第14行代码通过调用它的成员函数resumeTopActivityLocked来执行启动应用程序Launcher的操作。

Step 3：ActivityStack.resumeTopActivityLocked

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public class ActivityStack {
02      ......
03
04      final ActivityManagerService mService;
05      ......
06
07      final boolean resumeTopActivityLocked(ActivityRecord prev) {
08          // Find the first activity that is not finishing.
09          ActivityRecord next = topRunningActivityLocked(null);
10          ......
```

```
11
12          if (next == null) {
13              // There are no more activities!  Let's just start up the
14              // Launcher...
15              if (mMainStack) {
16                  return mService.startHomeActivityLocked();
17              }
18          }
19
20          ......
21      }
22
23      ......
24  }
```

第9行代码首先调用ActivityStack类的成员函数topRunningActivityLocked来获得当前Activity组件堆栈顶端的Activity组件。由于到目前为止，系统还没有启动过任何Activity组件，因此，这时候从Activity组件堆栈顶端获得的Activity组件就会等于null，即第12行的if语句会等于true。在这种情况下，接下来第15行到第17行代码就会将应用程序Launcher启动起来，以便它可以将系统中已经安装了的应用程序显示出来。

ActivityStack类的成员变量mMainStack是一个布尔变量，当它的值等于true时，就表示当前正在处理的一个ActivityStack对象是用来描述系统的Activity组件堆栈的。在这种情况下，第15行的if语句就会等于true，因此，接下来第16行代码就会调用ActivityStack类的成员变量mService的成员函数startHomeActivityLocked来启动应用程序Launcher。

由于ActivityStack类的成员变量mService指向了系统的Activity管理服务ActivityManagerService，即它的类型为ActivityManagerService，因此，接下来我们就继续分析ActivityManagerService类的成员函数startHomeActivityLocked的实现。

Step 4：ActivityManagerService.startHomeActivityLocked

frameworks/base/services/java/com/android/server/am/ActivityStack.java

```
01 public final class ActivityManagerService extends ActivityManagerNative
02          implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
03      ......
04
05      boolean startHomeActivityLocked() {
06          ......
07
08          Intent intent = new Intent(
09              mTopAction,
10              mTopData != null ? Uri.parse(mTopData) : null);
11          intent.setComponent(mTopComponent);
12          if (mFactoryTest != SystemServer.FACTORY_TEST_LOW_LEVEL) {
13              intent.addCategory(Intent.CATEGORY_HOME);
14          }
15          ActivityInfo aInfo =
16              intent.resolveActivityInfo(mContext.getPackageManager(),
17              STOCK_PM_FLAGS);
18          if (aInfo != null) {
19              intent.setComponent(new ComponentName(
20                  aInfo.applicationInfo.packageName, aInfo.name));
21              // Don't do this if the home app is currently being
22              // instrumented.
23              ProcessRecord app = getProcessRecordLocked(aInfo.processName,
24                  aInfo.applicationInfo.uid);
25              if (app == null || app.instrumentationClass == null) {
26                  intent.setFlags(intent.getFlags() | Intent.FLAG_ACTIVITY_NEW_TASK);
27                  mMainStack.startActivityLocked(null, intent, null, null, 0, aInfo,
```

```
28                      null, null, 0, 0, 0, false, false);
29              }
30          }
31
32          return true;
33      }
34
35      ......
36  }
```

ActivityManagerService类的成员变量mFactoryTest用来描述系统的运行模式。系统的运行模式有工厂测试模式和非工厂测试模式两种，其中，工厂测试模式又可以进一步划分为低级工厂测试模式和高级工厂测试模式两种。当系统分别运行在非工厂测试模式、低级工厂测试模式和高级工厂测试模式时，ActivityManagerService类的成员变量mFactoryTest的值就分别等于SystemServer.FACTORY_TEST_OFF、SystemServer.FACTORY_TEST_LOW_LEVEL和SystemServer. FACTORY_TEST_HIGH_LEVEL。

ActivityManagerService类的成员变量mTopAction、mTopData和mTopComponent是用来描述系统中第一个被启动的Activity组件的。在非工厂测试模式和高级工厂测试模式中，成员变量mTopAction的值等于Intent.ACTION_MAIN，而成员变量mTopData和mTopComponent的值均等于null；在低级工厂测试模式中，成员变量mTopAction和mTopData的值分别等于 Intent.ACTION_FACTORY_TEST和null，而成员变量mTopComponent用来描述一个Action名称等于Intent.ACTION_FACTORY_TEST的Activity组件的实现类名称。通过设置这三个成员变量的值，我们就可以让系统在不同的运行模式中启动一个不同的首个应用程序，这样就可以方便测试系统的功能。

我们假设系统是运行在非工厂测试模式中的，即假设ActivityManagerService类的成员变量mFactoryTest的值等于SystemServer.FACTORY_TEST_OFF。在这种情况下，第8行到第17行代码首先从Pakcage管理服务PakcageManagerService中获得一个类型和Action名称分别等于Intent.CATEGORY_HOME和Intent.ACTION_MAIN的Activity组件的信息，接着第23行和第24行代码再检查这个Activity组件所运行在的一个应用程序进程是否已经启动起来了。如果还没有启动起来，那么就说明系统中的第一个应用程序，即应用程序Launcher还没有启动起来。因此，最后第27行和第28行代码就会调用ActivityManagerService类的成员变量mMainStack的成员函数startActivityLocked来启动这个应用程序。

由于ActivityManagerService类的成员变量mMainStack指向了系统的Activity组件堆栈，即它的类型为ActivityStack，因此，接下来我们就继续分析ActivityStack类的成员函数startActivityLocked的实现。

Step 5：ActivityStack.startActivityLocked

第5步可以参考前面7.2小节中的第8步。从前面的调用过程可以知道，这一步是用来启动应用程序Launcher中一个类型和Action名称分别等于Intent.CATEGORY_HOME和Intent.ACTION_MAIN的Activity组件的。

在应用程序Launcher的配置文件AndroidManifest.xml中，Activity组件Launcher的类型和Action名称分别被设置为Intent.CATEGORY_HOME和Intent.ACTION_MAIN，如下所示。

packages/apps/Launcher2/AndroidManifest.xml

```
01 <manifest
02     xmlns:android="http://schemas.android.com/apk/res/android"
03     package="com.android.launcher"
04     android:sharedUserId="@string/sharedUserId"
05     >
06     ......
07     <application
08         android:name="com.android.launcher2.LauncherApplication"
09         android:process="@string/process"
10         android:label="@string/application_name"
```

```
11          android:icon="@drawable/ic_launcher_home">
12          <activity
13              android:name="com.android.launcher2.Launcher"
14              android:launchMode="singleTask"
15              android:clearTaskOnLaunch="true"
16              android:stateNotNeeded="true"
17              android:theme="@style/Theme"
18              android:screenOrientation="nosensor"
19              android:windowSoftInputMode="stateUnspecified|adjustPan">
20              <intent-filter>
21                  <action android:name="android.intent.action.MAIN" />
22                  <category android:name="android.intent.category.HOME" />
23                  <category android:name="android.intent.category.DEFAULT" />
24                  <category android:name="android.intent.category.MONKEY"/>
25                  </intent-filter>
26          </activity>
27          ......
28      </application>
29  </manifest>
```

因此，这一步执行完成之后，应用程序Launcher的Launcher组件就会被启动起来，即它的成员函数onCreate就会被调用。由于Launcher组件在启动过程中，会将系统中已经安装了的应用程序显示在系统屏幕中，因此，在接下来的第6步到第13步中，我们就从它的成员函数onCreate开始继续分析系统中已经安装了的应用程序的显示过程，如图16-5所示。

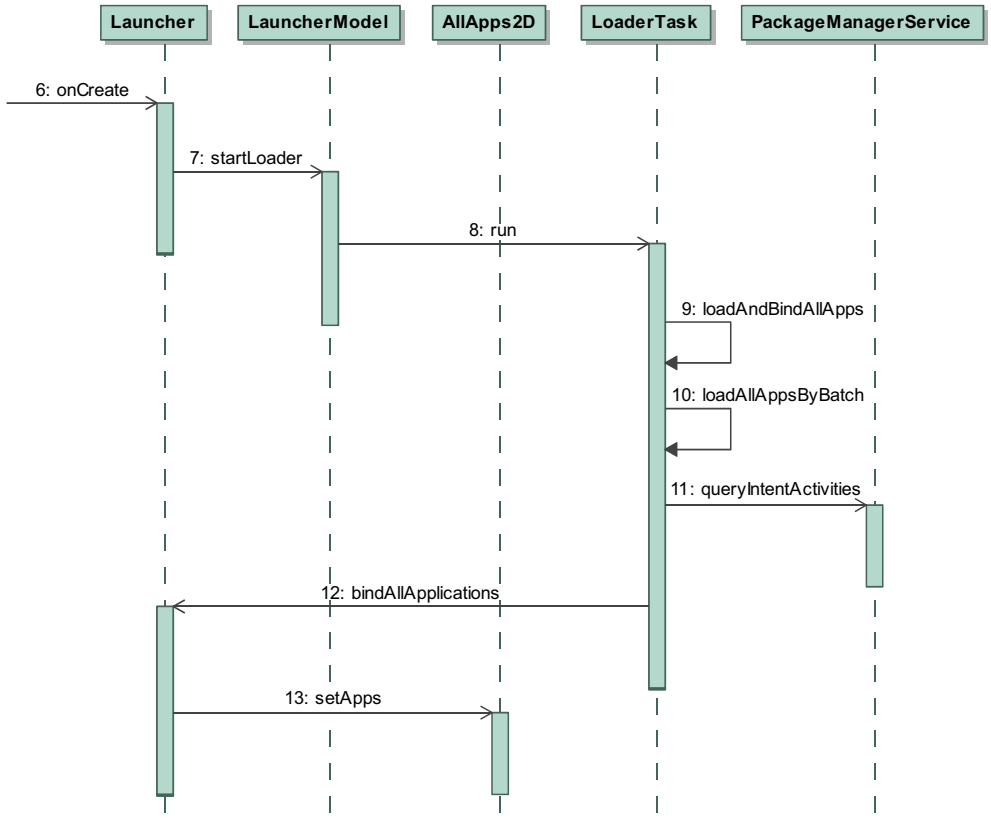


图16-5　应用程序的显示过程——Step 6至Step 13

Step 6：Launcher.onCreate

packages/apps/Launcher2/src/com/android/launcher2/Launcher.java

```
01 public final class Launcher extends Activity
02         implements View.OnClickListener, OnLongClickListener, LauncherModel.Callbacks, AllAppsView.Watcher {
```

```
03      ......
04
05      private Bundle mSavedState;
06      ......
07
08      private boolean mRestoring;
09      ......
10
11      private LauncherModel mModel;
12      ......
13
14      @Override
15      protected void onCreate(Bundle savedInstanceState) {
16          ......
17
18          LauncherApplication app = ((LauncherApplication)getApplication());
19          mModel = app.setLauncher(this);
20          ......
21
22          mSavedState = savedInstanceState;
23          restoreState(mSavedState);
24          ......
25
26          if (!mRestoring) {
27              mModel.startLoader(this, true);
28          }
29
30          ......
31      }
32
33      ......
34  }
```

第18行代码首先调用Launcher类的成员函数getApplication来获得一个用来描述应用程序Launcher的Application对象。

应用程序Launcher在它的配置文件AndroidManifest.xml中配置了一个自定义的Application对象，如下所示。

packages/apps/Launcher2/AndroidManifest.xml

```
01 <manifest
02      xmlns:android="http://schemas.android.com/apk/res/android"
03      package="com.android.launcher"
04      android:sharedUserId="@string/sharedUserId"
05      >
06      ......
07      <application
08          android:name="com.android.launcher2.LauncherApplication"
09          android:process="@string/process"
10          android:label="@string/application_name"
11          android:icon="@drawable/ic_launcher_home">
12          ......
13      </application>
14  </manifest>
```

从application标签的android:name属性可以知道，应用程序Launcher自定义的Application对象的类型为LauncherApplication，因此，Launcher类的成员函数onCreate可以将前面所获得的一个Application对象安全地转换为一个LauncherApplication对象，如前面第18行代码所示。

用来描述应用程序Launcher的一个LauncherApplication对象在创建时，会在内部创建一个LauncherModel对象，并且保存在它的成员变量mModel中，如下所示。

packages/apps/Launcher2/src/com/android/launcher2/LauncherApplication.java

```
01 public class LauncherApplication extends Application {
02     public LauncherModel mModel;
03     ......
04
05     @Override
06     public void onCreate() {
07         ......
08
09         mModel = new LauncherModel(this, mIconCache);
10         ......
11
12     }
13
14     ......
15 }
```

返回到Launcher类的成员函数onCreate中，接下来第19行代码会调用LauncherApplication类的成员函数setLauncher将当前正在启动的Launcher组件设置到前面所获得的一个LauncherApplication对象中，如下所示。

packages/apps/Launcher2/src/com/android/launcher2/LauncherApplication.java

```
01 public class LauncherApplication extends Application {
02     public LauncherModel mModel;
03     ......
04
05     LauncherModel setLauncher(Launcher launcher) {
06         mModel.initialize(launcher);
07         return mModel;
08     }
09
10     ......
11 }
```

第6行代码使用当前正在启动的Launcher组件来初始化LauncherApplication类的成员变量mModel所描述的一个LauncherModel对象，这是通过调用LauncherModel类的成员函数initialize来实现的，如下所示。

packages/apps/Launcher2/src/com/android/launcher2/LauncherModel.java

```
01 public class LauncherModel extends BroadcastReceiver {
02     ......
03
04     private WeakReference<Callbacks> mCallbacks;
05     ......
06
07     public void initialize(Callbacks callbacks) {
08         synchronized (mLock) {
09             mCallbacks = new WeakReference<Callbacks>(callbacks);
10         }
11     }
12
13     ......
14 }
```

从前面的调用过程可以知道，参数callbacks指向了当前正在启动的Launcher组件的一个Callbacks接口，第9行代码首先将这个参数封装成一个弱引用对象，接着再将这个弱引用对象保存在LauncherModel类的成员变量mCallbacks中，用来描述当前正在启动的Launcher组件。

返回到LauncherApplication类的成员函数setLauncher中，接下来第7行代码就会将前面已经初始化完成的一个LauncherModel对象返回给调用者，即Launcher类的成员函数onCreate。

返回到Launcher类的成员函数onCreate中，接下来第19行代码会将前面所获得的一个LauncherModel

对象保存在Launcher类的成员变量mModel中。在接下来的内容中，我们就会看到，这个LauncherModel对象是用来加载系统中已经安装了的应用程序的信息的。

在Launcher类的成员函数onCreate中，参数savedInstanceState用来描述当前正在启动的Launcher组件上一次被销毁之前的状态数据。第22行代码首先将它保存在Launcher类的成员变量mSavedState中，接着第23行代码再调用Launcher类的成员函数restoreState来恢复当前正在启动的Launcher组件上一次被销毁之前的状态数据。

Launcher类的成员函数restoreState在恢复当前正在启动的Launcher组件上一次被销毁之前的状态数据的过程中，会将Launcher类的成员变量mRestoring的值设置为false，表示Launcher类的成员函数onCreate需要马上加载系统中已经安装了的应用程序的信息。在这种情况下，第27行代码就会调用前面所获得的一个LauncherModel对象的成员函数startLoader，以便可以加载系统中已经安装了的应用程序的信息。

Step 7：LauncherModel.startLoader

packages/apps/Launcher2/src/com/android/launcher2/LauncherModel.java

```
01 public class LauncherModel extends BroadcastReceiver {
02     ......
03
04     private LoaderTask mLoaderTask;
05     ......
06
07     private static final HandlerThread sWorkerThread = new HandlerThread("launcher-loader");
08     static {
09         sWorkerThread.start();
10     }
11     private static final Handler sWorker = new Handler(sWorkerThread.getLooper());
12     ......
13
14     public void startLoader(Context context, boolean isLaunching) {
15         synchronized (mLock) {
16             ......
17
18             // Don't bother to start the thread if we know it's not going to do anything
19             if (mCallbacks != null && mCallbacks.get() != null) {
20                 // If there is already one running, tell it to stop.
21                 LoaderTask oldTask = mLoaderTask;
22                 if (oldTask != null) {
23                     if (oldTask.isLaunching()) {
24                         // don't downgrade isLaunching if we're already running
25                         isLaunching = true;
26                     }
27                     oldTask.stopLocked();
28                 }
29                 mLoaderTask = new LoaderTask(context, isLaunching);
30                 sWorker.post(mLoaderTask);
31             }
32         }
33     }
34
35     ......
36 }
```

LauncherModel类有一个类型为LoaderTask的成员变量mLoaderTask，它用来执行加载系统中已经安装了的应用程序的信息的操作。由于加载系统中已经安装了的应用程序的信息是一个比较耗时的操作，因此，LauncherModel类将它放在一个后台线程中执行。这个后台线程是使用LauncherModel类的静态成员变量sWorkerThread来描述的。

从第7行到第10行代码可以知道，LauncherModel类的静态成员变量sWorkerThread指向了一个HandlerThread对象。从前面15.2小节的内容又可以知道，一个HandlerThread对象所描述的一个线程是

具有消息循环的。为什么LauncherModel类需要使用一个带有消息循环的线程来执行加载系统中已经安装了的应用程序的信息的操作呢?

我们知道，应用程序Launcher在运行过程中，系统中的应用程序是会发生变化的，即系统可能会增加新的应用程序，或者删除原来已有的应用程序。因此，应用程序Launcher需要经常执行加载系统中已经安装了的应用程序的信息的操作，以便可以实时地反映出系统当前已经安装了的应用程序的情况。为了避免重复地创建新的线程来执行加载系统中已经安装了的应用程序的信息的操作，LauncherModel类就创建了一个带有消息循环的线程。每当应用程序Launcher需要重新显示系统中已经安装了的应用程序时，LauncherModel类的成员函数startLoader就会向这个线程的消息队列发送一个类型为LoaderTask的消息，以便它可以重新加载系统中已经安装了的应用程序的信息。

从第11行代码可以看出，LauncherModel类的静态成员变量sWorker指向了一个Handler对象，这个Handler对象是用来向LauncherModel类的静态成员变量sWorkerThread所描述的一个线程的消息队列发送消息的。

了解了LauncherModel类的成员变量mLoaderTask，以及静态成员变量sWorkerThread和sWorker的作用之后，接下来我们就可以分析LauncherModel类的成员函数startLoader的实现了。

从前面的Step 6可以知道，LauncherModel类的成员变量mCallbacks引用了Launcher组件的一个Callbacks接口，它的值不等于null，即第19行的if语句等于true，接下来第21行到第30行代码就会执行加载系统中已经安装了的应用程序的信息的操作。

第21行到第28行代码首先检查LauncherModel类是否正在执行加载系统中已经安装了的应用程序的信息的操作。如果是，那么就会先停止这个操作；接着第29行和第30行代码再通过LauncherModel类的静态成员变量sWorker所描述的一个Handler对象，将一个新创建的LoaderTask对象封装成一个消息发送到LauncherModel类的静态成员变量sWorkerThread所描述的一个线程的消息队列中。

LoaderTask类实现了Runnable接口，当它所描述的一个消息被处理时，它的成员函数run就会被调用。因此，接下来我们就继续分析LoaderTask类的成员函数run的实现，以便可以了解它是如何加载系统中已经安装了的应用程序的信息的。

Step 8：LoaderTask.run

packages/apps/Launcher2/src/com/android/launcher2/LauncherModel.java

```
01 public class LauncherModel extends BroadcastReceiver {
02     ......
03
04     private class LoaderTask implements Runnable {
05         ......
06
07         public void run() {
08             ......
09
10             final Callbacks cbk = mCallbacks.get();
11             final boolean loadWorkspaceFirst = cbk != null ? (!cbk.isAllAppsVisible()) : true;
12
13             keep_running: {
14                 ......
15
16                 if (loadWorkspaceFirst) {
17                     ......
18                     loadAndBindWorkspace();
19                 } else {
20                     ......
21                     loadAndBindAllApps();
22                 }
23
```

```
24                  if (mStopped) {
25                      break keep_running;
26                  }
27
28                  // second step
29                  if (loadWorkspaceFirst) {
30                      ......
31                      loadAndBindAllApps();
32                  } else {
33                      ......
34                      loadAndBindWorkspace();
35                  }
36              }
37
38              ......
39          }
40
41          ......
42      }
43
44      ......
45  }
```

应用程序Launcher按照工作区（workspace）的形式来显示系统中已经安装了的应用程序的快捷图标。每一个工作区都是用来描述一个抽象桌面的，它由*N*个屏幕组成。工作区中的每一个屏幕由*X* * *Y*个单元格组成。屏幕中的每一个单元格用来显示一个应用程序快捷图标。用户可以设置当前要显示的屏幕，同时也可以将一个应用程序快捷图标从一个单元格移动到另外一个单元格。

第10行和第11行代码首先判断显示在Launcher组件用户界面上的应用程序快捷图标是否可见。如果可见，那么第16行到第35行代码就会首先调用LoaderTask类的成员函数loadAndBindAllApps来加载系统中已经安装了的应用程序的信息，接着再调用LoaderTask类的成员函数loadAndBindWorkspace来加载工作区信息；否则，第16行到第35行代码就会首先调用LoaderTask类的成员函数loadAndBindWorkspace来加载工作区信息，接着再调用LoaderTask类的成员函数loadAndBindAllApps来加载系统中已经安装了的应用程序的信息。

我们只关注系统中已经安装了的应用程序的信息的加载过程，因此，接下来我们就继续分析LoaderTask类的成员函数loadAndBindAllApps的实现。

Step 9：LoaderTask.loadAndBindAllApps

packages/apps/Launcher2/src/com/android/launcher2/LauncherModel.java

```
01 public class LauncherModel extends BroadcastReceiver {
02     ......
03
04     private boolean mAllAppsLoaded;
05     ......
06
07     private class LoaderTask implements Runnable {
08         ......
09
10         private void loadAndBindAllApps() {
11             ......
12
13             if (!mAllAppsLoaded) {
14                 loadAllAppsByBatch();
15                 if (mStopped) {
16                     return;
17                 }
18                 mAllAppsLoaded = true;
```

```
19              } else {
20                  onlyBindAllApps();
21              }
22          }
23
24          ......
25      }
26
27      ......
28  }
```

第13行的if语句首先检查LauncherModel类是否已经加载过系统中已经安装了的应用程序的信息。如果已经加载过，那么LauncherModel类的成员变量mAllAppsLoaded的值就会等于true，接着第20行代码就会直接调用LoaderTask类的成员函数onlyBindAllApps将这些已经加载了的应用程序的信息显示在屏幕中；否则，第14行代码就会调用LoaderTask类的成员函数loadAllAppsByBatch首先加载系统中已经安装了的应用程序的信息，接着再将这些已经加载了的应用程序的信息显示在屏幕中。

在我们这个场景中，由于Launcher组件是刚刚启动起来的，即它还没有加载过系统中已经安装了的应用程序的信息，因此，接下来我们就继续分析LoaderTask类的成员函数loadAllAppsByBatch的实现，以便可以了解它是如何加载以及显示系统中已经安装了的应用程序的信息的。

Step 10：LoaderTask.loadAllAppsByBatch

packages/apps/Launcher2/src/com/android/launcher2/LauncherModel.java

```
01 public class LauncherModel extends BroadcastReceiver {
02     ......
03
04     private int mBatchSize; // 0 is all apps at once
05     ......
06
07     private DeferredHandler mHandler = new DeferredHandler();
08     ......
09
10     private AllAppsList mAllAppsList; // only access in worker thread
11     ......
12
13     private class LoaderTask implements Runnable {
14         ......
15
16         private void loadAllAppsByBatch() {
17             ......
18
19             final Callbacks oldCallbacks = mCallbacks.get();
20             ......
21
22             final Intent mainIntent = new Intent(Intent.ACTION_MAIN, null);
23             mainIntent.addCategory(Intent.CATEGORY_LAUNCHER);
24
25             final PackageManager packageManager = mContext.getPackageManager();
26             List<ResolveInfo> apps = null;
27
28             int N = Integer.MAX_VALUE;
29             ......
30
31             int i=0;
32             int batchSize = -1;
33             while (i < N && !mStopped) {
34                 if (i == 0) {
35                     mAllAppsList.clear();
36                     ......
37                     apps = packageManager.queryIntentActivities(mainIntent, 0);
38                     ......
39
```

```
40                      N = apps.size();
41                      ......
42
43                      if (mBatchSize == 0) {
44                          batchSize = N;
45                      } else {
46                          batchSize = mBatchSize;
47                      }
48                      ......
49
50                      Collections.sort(apps,
51                          new ResolveInfo.DisplayNameComparator(packageManager));
52                  }
53                  ......
54
55                  for (int j=0; i<N && j<batchSize; j++) {
56                      // This builds the icon bitmaps.
57                      mAllAppsList.add(new ApplicationInfo(apps.get(i), mIconCache));
58                      i++;
59                  }
60
61                  final boolean first = i <= batchSize;
62                  final Callbacks callbacks = tryGetCallbacks(oldCallbacks);
63                  final ArrayList<ApplicationInfo> added = mAllAppsList.added;
64                  mAllAppsList.added = new ArrayList<ApplicationInfo>();
65
66                  mHandler.post(new Runnable() {
67                      public void run() {
68                          ......
69                          if (callbacks != null) {
70                              if (first) {
71                                  callbacks.bindAllApplications(added);
72                              } else {
73                                  callbacks.bindAppsAdded(added);
74                              }
75                              ......
76                          }
77                          ......
78                      }
79                  });
80
81                  ......
82              }
83
84              ......
85          }
86
87          ......
88      }
89
90      ......
91  }
```

第22行和第23行代码首先创建了一个类型和Action名称分别等于Intent.CATEGORY_LAUNCHER和Intent.ACTION_MAIN的Intent对象mainIntent，然后第25行代码再获得一个PackageManagerService代理对象packageManager，以便接下来第33行到第82行的while循环可以请求Package管理服务PackageManagerService返回系统中已经安装了的应用程序的根Activity组件信息。

第33行到第82行的while循环以分批的形式来显示系统中已经安装了的应用程序的根Activity组件，其中，变量i和变量batchSize分别用来描述下一批要显示的根Activity组件的位置以及个数。

第34行的if语句检查变量i的值是否等于0。如果等于0，那么就说明目前需要显示的是第一批根Activity组件。在这种情况下，由于函数还没有请求Package管理服务PackageManagerService返回系统中

已经安装了的应用程序的根Activity组件信息，因此，第37行代码就会首先通过PackageManagerService代理对象packageManager的成员函数queryIntentActivities来获得这些根Activity组件信息，并且将它们保存在ResolveInfo列表apps中。接着第43行的if语句再检查LauncherModel类的成员变量mBatchSize的值是否等于0。如果等于0，那么就说明需要一次性地将所有的根Activity组件都显示出来，因此，第44行代码就会将变量batchSize的值设置为N，即前面所获得的根Activity组件的总数；否则，第46行代码就会使用LauncherModel类的成员变量mBatchSize来初始化变量batchSize。最后第50行和第51行代码调用Collections类的静态成员函数sort来对保存在ResolveInfo列表apps中的根Activity组件进行排序，即按照这些根Activity组件的显示名称从小到大的顺序进行排序。

LauncherModel类的成员变量mAllAppsList指向了一个AllAppsList对象。这个AllAppsList对象内部有一个类型为ApplicationInfo的列表added，用来临时保存当前需要显示的根Activity组件。第55行到第59行的for循环依次将当前需要显示的根Activity组件封装成ApplicationInfo对象，并且添加到这个临时列表中。

第61行代码检查当前需要显示的根Activity组件是否是第一批根Activity组件。如果是，那么变量first的值就会被设置为true。第62行代码调用LoaderTask类的成员函数tryGetCallbacks来获得LauncherModel类的成员变量mCallbacks所描述的一个Callbacks接口，并且保存在变量callbacks中。从前面Step 6的内容可以知道，这个Callbacks接口指向了应用程序Launcher中的Launcher组件。第63行和第64行代码首先将当前需要显示的根Activity组件转移到另外一个ApplicationInfo列表added中，接着再将LauncherModel类的成员变量mAllAppsList所指向的一个AllAppsList对象内部的一个临时ApplicationInfo列表added清空，即让这个临时列表指向一个新创建的ApplicationInfo列表，以便接下来可以将下一批需要显示的根Activity组件保存在里面。

LauncherModel类的成员变量mHandler指向了一个类型为DeferredHandler的Handler对象，这个Handler对象是在应用程序Launcher的主线程中创建的，因此，它可以用来向应用程序Launcher的主线程的消息队列发送消息。DeferredHandler类在内部有一个自己的消息队列，用来临时保存需要发送到应用程序Launcher的主线程的消息队列的消息。

第66行到第79行代码首先创建了一个Runnable对象，接着再将这个Runnable对象封装成一个消息发送到应用程序Launcher的主线程的消息队列中。当这个消息得到处理时，第69行到第76行代码就会调用前面所获得的一个Callbacks接口callbacks的成员函数bindAllApplications或者bindAppsAdded，来显示保存在临时ApplicationInfo列表added中的根Activity组件，这取决于变量first的值是等于true还是false。Callbacks接口callbacks的成员函数bindAllApplications在将指定的根Activity组件显示在屏幕上之前，首先会将原来已经在屏幕上显示的根Activity组件清空，而成员函数bindAppsAdded只是将指定的根Activity组件增加到屏幕上来。这两个成员函数的实现是类似的，接下来我们重点分析前一个成员函数的实现。

前面提到，Callbacks接口callbacks实际指向的是应用程序Launcher中的Launcher组件，因此，第71行代码实际上是调用Launcher组件的成员函数bindAllApplications来显示保存在临时ApplicationInfo列表added中的根Activity组件。在继续分析Launcher组件的成员函数bindAllApplications的实现之前，我们首先分析PackageManagerService类的成员函数queryIntentActivities的实现，以便可以了解前面第37行代码是如何获得系统中已经安装了的应用程序的根Activity组件的信息的。

Step 11：PackageManagerService.queryIntentActivities

frameworks/base/services/java/com/android/server/PackageManagerService.java

```
01 class PackageManagerService extends IPackageManager.Stub {
02     ......
03
```

```
04      // All available activities, for your resolving pleasure.
05      final ActivityIntentResolver mActivities =
06              new ActivityIntentResolver();
07      ......
08
09      public List<ResolveInfo> queryIntentActivities(Intent intent,
10              String resolvedType, int flags) {
11          ......
12
13          synchronized (mPackages) {
14              String pkgName = intent.getPackage();
15              if (pkgName == null) {
16                  return (List<ResolveInfo>)mActivities.queryIntent(intent,
17                          resolvedType, flags);
18              }
19
20              ......
21          }
22
23          ......
24      }
25
26      ......
27  }
```

从前面16.1小节的Step 11可以知道，系统中已经安装了的应用程序的Activity组件都保存在PackageManagerService类的成员变量mActivities中，因此，第13行到第21行代码就可以通过这个成员变量来获得指定的Activity组件。

从前面Step 10的内容可以知道，参数intent所描述的一个Intent对象并没有指定Package名称，即第15行的if语句为true，接下来第16行和第17行代码就会调用PackageManagerService类的成员变量mActivities的成员函数queryIntent来获得指定的Activity组件。

从前面Step 10的内容还可以知道，参数intent所描述的一个Intent对象的类型和Action名称分别被指定为Intent.CATEGORY_LAUNCHER和Intent.ACTION_MAIN，因此，第16行和第17行代码所获得的Activity组件即为系统中已经安装了的应用程序的根Activity组件。这些根Activity组件最终被保存在一个ResolveInfo列表中返回给调用者。

这一步执行完成之后，返回到前面的Step 10中，即LoaderTask类的成员函数loadAllAppsByBatch中，接下来就会调用Launcher组件的成员函数bindAllApplications来显示系统中已经安装了的应用程序的根Activity组件。

Step 12：Launcher.bindAllApplications

packages/apps/Launcher2/src/com/android/launcher2/Launcher.java

```
01 public final class Launcher extends Activity
02          implements View.OnClickListener, OnLongClickListener, LauncherModel.Callbacks, AllAppsView.Watcher {
03      ......
04
05      private AllAppsView mAllAppsGrid;
06      ......
07
08      public void bindAllApplications(ArrayList<ApplicationInfo> apps) {
09          mAllAppsGrid.setApps(apps);
10      }
11
12      ......
13  }
```

Launcher类的成员变量mAllAppsGrid指向了一个AllApps2D对象或者一个AllApps3D对象，这取决

于Launcher组件采用的是2D界面还是3D界面。假设Launcher组件采用的是2D界面，那么第9行代码就会调用AllApps2D类的成员函数setApps将保存在ApplicationInfo列表apps中的根Activity组件显示到屏幕上来。

Step 13：AllApps2D.setApps

packages/apps/Launcher2/src/com/android/launcher2/AllApps2D.java

```
01 public class AllApps2D
02     extends RelativeLayout
03     implements AllAppsView,
04         AdapterView.OnItemClickListener,
05         AdapterView.OnItemLongClickListener,
06         View.OnKeyListener,
07         DragSource {
08     ......
09
10     private ArrayList<ApplicationInfo> mAllAppsList = new ArrayList<ApplicationInfo>();
11     ......
12
13     public void setApps(ArrayList<ApplicationInfo> list) {
14         mAllAppsList.clear();
15         addApps(list);
16     }
17
18     ......
19 }
```

AllApps2D类的成员变量mAllAppsList用来保存所有需要在屏幕上显示的根Activity组件，第14行代码首先清空它的内容，接着第15行代码再调用AllApps2D类的成员函数addApps将保存在ApplicationInfo列表list中的根Activity组件显示到屏幕上来。

AllApps2D类的成员函数addApps的实现如下所示。

packages/apps/Launcher2/src/com/android/launcher2/AllApps2D.java

```
01 public class AllApps2D
02     extends RelativeLayout
03     implements AllAppsView,
04         AdapterView.OnItemClickListener,
05         AdapterView.OnItemLongClickListener,
06         View.OnKeyListener,
07         DragSource {
08     ......
09
10     private GridView mGrid;
11     ......
12
13     private AppsAdapter mAppsAdapter;
14     ......
15
16     public void addApps(ArrayList<ApplicationInfo> list) {
17         final int N = list.size();
18
19         for (int i=0; i<N; i++) {
20             final ApplicationInfo item = list.get(i);
21             int index = Collections.binarySearch(mAllAppsList, item,
22                 LauncherModel.APP_NAME_COMPARATOR);
23             if (index < 0) {
24                 index = -(index+1);
25             }
26             mAllAppsList.add(index, item);
27         }
```

```
28          mAppsAdapter.notifyDataSetChanged();
29      }
30
31      ......
32  }
```

AllApps2D类的成员变量mGrid指向了一个GridView控件，它用来显示系统中已经安装了的应用程序的快捷图标，这些应用程序快捷图标的信息就保存在AllApps2D类的成员变量mAllAppsList中。此外，AllApps2D类的成员变量mGrid所指向的一个GridView控件还关联着一个数据适配器，这个数据适配器是通过AllApps2D类的成员变量mAppsAdapter来描述的。

每一个应用程序快捷图标的信息都是使用一个ApplicationInfo对象来描述的，这些ApplicationInfo对象按照它们所描述的根Activity组件的显示名称从小到大的顺序保存在AllApps2D类的成员变量mAllAppsList中。因此，第19行到第27行的for循环在将保存在列表list中的ApplicationInfo对象添加到AllApps2D类的成员变量mAllAppsList中之前，首先要找到它们在AllApps2D类的成员变量mAllAppsList中的位置，这是通过调用Collections类的成员函数binarySearch来实现的。

将保存在列表list中的所有ApplicationInfo对象都添加到AllApps2D类的成员变量mAllAppsList中之后，最后第28行代码就调用AllApps2D类的成员变量mAppsAdapter所描述的一个数据适配器的成员函数notifyDataSetChanged来更新屏幕上的应用程序快捷图标。

这一步执行完成之后，系统中已经安装了的应用程序的根Activity组件就以快捷图标的形式显示在屏幕上了。当用户点击这些快捷图标时，AllApps2D类的成员函数onItemClick就会被调用，调用的结果就是将对应的应用程序的根Activity组件启动起来。

AllApps2D类的成员函数onItemClick的实现如下所示。

packages/apps/Launcher2/src/com/android/launcher2/AllApps2D.java

```
01 public class AllApps2D
02     extends RelativeLayout
03     implements AllAppsView,
04         AdapterView.OnItemClickListener,
05         AdapterView.OnItemLongClickListener,
06         View.OnKeyListener,
07         DragSource {
08     ......
09
10     private Launcher mLauncher;
11     ......
12
13     public void onItemClick(AdapterView parent, View v, int position, long id) {
14         ApplicationInfo app = (ApplicationInfo) parent.getItemAtPosition(position);
15         mLauncher.startActivitySafely(app.intent, app);
16     }
17
18     ......
19 }
```

第14行代码首先获得用来描述被点击的快捷图标的一个ApplicationInfo对象。这个ApplicationInfo对象的成员变量intent指向了一个Intent对象，它是用来启动与被点击的快捷图标所关联的一个应用程序的根Activity组件的。

AllApps2D类的成员变量mLauncher指向了应用程序Launcher中的Launcher组件，最后第15行代码就调用它的成员函数startActivitySafely来启动指定的应用程序的根Activity组件，这样就可以将一个指定的应用程序启动起来，如前面7.2小节的内容所示。

至此，系统中已经安装了的应用程序的显示过程就分析完成了。